КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет інформаційних технологій

Кафедра технологій управління

Спеціальність 122 - Комп'ютерні науки,
програма "Інформаційна аналітика та впливи"

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему:

«Аналіз підходів до класифікації аудіофайлів методами машинного навчання»

Студента 2-го курсу групи ІАВ-21

Кучерявого Ярослава Анатолійовича
(прізвище, ім'я, по батькові)

(підпис студента)

Науковий керівник:

доктор технічних наук, доцент
(науковий ступінь, вчене звання)

Єгорченков Олексій Володимирович
(прізвище, ім'я, по батькові)

(дата) *(підпис)*

Попередній захист:

(Висновок: "До захисту в Державній екзаменаційній комісії")

Завідувач кафедри
технологій управління

(підпис) *(прізвище, ініціали)* *(дата)*

Київ – 2022

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет інформаційних технологій

Кафедра технологій управління
Освітньо-кваліфікаційний рівень Магістр
Спеціальність 122 - Комп'ютерні науки
Програма Інформаційна аналітика та впливи

ЗАТВЕРДЖУЮ
Завідувач кафедри
професор Морозов В.В.
____
"____" _________20___року

З А В Д А Н Н Я
НА ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Студент Кучерявий Ярослав Анатолійович

Група ІАВ-21

1. Тема кваліфікаційної роботи
Аналіз підходів до класифікації аудіофайлів методами машинного навчання

Затверджена наказом по від "___"______20__ р. № ___.

2. **Строк подання студентом готової роботи** - "___"_______ 20___ р.

3. Цільова установка та вихідні дані до роботи зібрати файли для класифікації власноруч, розробити датасет їх метрик, для порівняльної класифікації використати датасет «SpotifyFeatures.csv», побудувати декілька моделей класифікаці та провести їх порівняння і аналіз.

4. Зміст роботи визначення предметної області завдання, аналіз теоретичних засад цифрової обробки звуку, пошук засобів вилучення акустичних метрик, розробка алгоритму переведення аудіофайлу у табличний вигляд за релевантними показниками, розробка алгоритму збору звукових файлів з веб-джерел, проведення нормалізації зібраних та контрольних даних, теоретичний аналіз алгоритмів класифікації, розробка, навчання та вилучення результатів передбачальних моделей класифікації, порівняння та пояснення отриманих результатів.

5. Перелік графічного матеріалу (слайдів) 145 рисунків, 1 формула, 1 таблиця, 51 слайд презентації доповіді.

6. Календарний план виконання роботи:

№ п/п	Назва частин роботи	%	Виконання роботи	
			За планом	Фактично
1.	Вибір теми дипломної роботи	2	01.10.2021	01.10.2021
2.	Протокол кафедри ТУ про затвердження тем дипломних робіт та призначення наукових керівників	1	09.11.2021	09.11.2021
3.	Формування переліку нормативних матеріалів, літератури з проблематики кваліфікаційної роботи	4	08.01.2022	09.01.2022
4.	Складання розгорнутого плану кваліфікаційної роботи	4	18.01.2022	15.01.2022
5.	Ознайомлення наукового керівника з розгорнутим планом кваліфікаційної роботи. Внесення змін.	2	19.01.2022 - 20.01.2022	18.01.2022
6.	Підготовка розділу 1 «Теоретичні засади та підходи для вирішення задачі аналізу музичної індустрії»	15	12.02.2022	15.02.2022
7.	Підготовка розділу 2 «Чисельне представлення аудіосигналів. Збір, оптимізація та нормалізація даних»	30	08.03.2022	13.03.2022
8.	Підготовка розділу 3 «Навчання та порівняння моделей класифікації різних видів»	30	16.03.2022	20.03.2022
9.	Оформлення кваліфікаційної роботи. Підготовка висновків і пропозицій	5	29.04.2022	29.04.2022
10.	Передача кваліфікаційної роботи науковому керівникові	1	04.05.2022	04.05.2022
11.	Передача кваліфікаційної роботи рецензенту для рецензування	1	11.05.2022	11.05.2022
12.	Попередній захист кваліфікаційної роботи	5	17.05.2022	17.05.2022

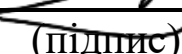
Дата видачі завдання “____” _________________ 20__ р.

Керівник роботи доктор технічних наук, доцент Єгорченков Олексій Володимирович
(посада, прізвище, ім’я, по батькові)

____
(підпис)

Завдання прийняв до виконання студент групи ІАВ-21

Кучерявий Ярослав Анатолійович
(прізвище, ім’я, по батькові)


____
(підпис)

ЗМІСТ

АНОТАЦІЯ ... 6

ВСТУП ... 8

РОЗДІЛ 1 ТЕОРЕТИЧНІ ЗАСАДИ ТА ПІДХОДИ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ АНАЛІЗУ МУЗИЧНОЇ ІНДУСТРІЇ ... 11

1.1 Цифрове представлення звуку ... 11

1.2. Сутність та задачі аудіоаналізу ... 11

1.3. Інструменти аудіоаналізу в Python ... 12

1.4. Музичні властивості в librosa ... 14

РОЗДІЛ 2. ЧИСЕЛЬНЕ ПРЕДСТАВЛЕННЯ АУДІОСИГНАЛІВ. ЗБІР, ОПТИМІЗАЦІЯ ТА НОРМАЛІЗАЦІЯ ДАНИХ ... 22

2.1. Розробка механізму представлення аудіофайлу ... 22

2.2 Зменшення розмірності даних для коректного порівняння моделей ... 27

2.3 Збір даних для навчання ... 33

2.4 Збір даних для порівняльної моделей. Остаточна нормалізація датасетів ... 49

РОЗДІЛ 3. НАВЧАННЯ ТА ПОРІВНЯННЯ МОДЕЛЕЙ КЛАСИФІКАЦІЇ РІЗНИХ ВИДІВ ... 54

3.1. Метод найближчих сусідів ... 54

3.2. Метод опорних векторів ... 62

3.3. Метод опорних векторів РБФ ... 70

3.4. Гаусові процеси ... 76

3.5. Дерево рішень ... 80

3.6. Випадковий ліс ... 86

3.7. Нейронна мережа (багатошаровий перцептрон) ... 92

3.8. AdaBoost ... 98

3.9. Наївний баєсів класифікатор 100
3.10. Квадратичний дискримінантний аналіз.................................... 102
3.11. Аналіз результатів.. 104
ВИСНОВОК 111
ПЕРЕЛІК ПОСИЛАНЬ 113

АНОТАЦІЯ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет інформаційних технологій
Кафедра технологій управління
Спеціальність 122 - Комп’ютерні науки,
освітня програма “Інформаційна аналітика та впливи”

Дипломна робота магістра Кучерявого Ярослава Анатолійовича.

Тема роботи – «Аналіз підходів до класифікації аудіофайлів методами машинного навчання».

Мета дипломної роботи магістра – Здійснення аналізу підходів до класифікації аудіофайлів методами машинного навчання.

Об’єкт дослідження – процеси класифікації аудіофайлів та акустичні властивості сигналів.

Предмет дослідження – інформаційні засоби, методи, моделі управління даними і процесами при аналізі акустичних властивостей сигналів і класифікації аудіофайлів.

Наукова новизна роботи полягає у розробці набору метрик для класифікації музики, створенні та порівнянні ефективності моделей жанрової класифікації та рекомендаціях щодо практичного застосування напрацювань.

Практична значимість роботи полягає у виявленні оптимальної архітектури, стеку технологій та дизайнерських рішень для розробки системи аналізу музичної індустрії.

В рамках роботи проведено аналіз теоретичних засад цифрової обробки звуку. Було вибрано оптимальний програмний засіб вилучення акустичних метрик сигналів, та розроблений алгоритм переведення аудіофайлу у табличний вигляд за релевантними показниками. Додатково було розроблено засіб збору аудіофайлів з веб-джерел, знайдено набір контрольних даних, проведена нормалізація зібраних та контрольних даних з 523 пісень, розбитих 6 жанрів як ціль класифікації.

Для побудови моделей класифікації був проведений теоретичний аналіз використовуваних алгоритмів класифікації. За його результатами були виконані розробка, навчання та вилучення результатів 10 передбачальних моделей на різних моделей. В результаті порівняння та пояснення отриманих результатів було розроблено рекомендації для оптимального підходу класифікації з використанням всіх отриманих напрацювань.

Дипломна робота складається зі вступу, основної частини, яка включає три розділи, висновків та списку використаних джерел. Всього основного тексту налічує 118 сторінок та перелік посилань з 54 джерел на 6 сторінках.

Ключові слова: цифрова обробка звуку, акустичні властивості, веб-скрапинг, нормалізація, класифікація, машинне навчання, багатошаровий перцептрон.

ВСТУП

Незважаючи на те, що на сьогоднішній день задачі Machine Learning переважно лежать в площині комп'ютерного зору та розпізнавання тексту, аналіз аудіосигналів також є однією зі спеціалізованих задач машинного навчання [1].

Визначальна властивість даної задачі полягає в тому, що аудіосигнал в закодованому цифровому представлені являє собою набір даних великого об'єму та складної структури. При опрацюванні аудіосигналів над цими даними виконуються математичні перетворення, які дозволяють виділяти окремі їх властивості, суттєві для кожного конкретного випадку. В залежності від задачі, властивості можуть бути як суто фізичними, так і пов'язаними зі сприйняттям сигналу людиною, зокрема, в рамках теорії музики [2]. Часто вилучені ознаки можуть бути визначені як великі дані (в статистичному сенсі), часто багатовимірні, які для корисного застосування вимагають програмної обробки. Таким чином, для вилучення корисних властивостей сигналів стають в нагоді Data Science та машинне навчання [3].

Практичне застосування засобів Machine Learning в рамках обробки звуку може полягати у розробці систем комунікації, розпізнання мови, створення та класифікації музики. Творчий підхід до застосування напрацювань у сфері аудіоаналізу може мати місце в багатьох сферах, не зв'язаних з акустикою безпосередньо: наприклад, в медицині або промисловості.

Завдання класифікації та кластеризації — одні з основних задач машинного навчання — знаходять широкого застосування в рамках аудіоаналізу. Так, в індустрії музики вони можуть використовуватись для поділу на жанри або інші категорії, аналізу вподобань аудиторії та прогнозування популярності пісень або виконавців.

Актуальність дослідження музичної індустрії є іманентною їй самій, оскільки підходи до написання музики перманентно розвиваються і

змінюються. Засоби цифрового перетворення та розпізнання звуку за визначенням спираються на математичний та акустичний апарат, тому їх розвиток має високе теоретичне значення. Використання ж сучасних засобів Machine Learning та Data Science дозволяє обробляти небувало великі об'єми даних, часто — з високою швидкістю, отримуючи нові результати вищої якості.

Мета дослідження:

Здійснення аналізу підходів до класифікації аудіофайлів методами машинного навчання.

Завдання дослідження:

- Дослідити сучасні засоби розпізнання звуку та аналізу аудіосигналів.
- Визначити можливості та перспективи аналізу аудіофайлів засобами Machine Learning та Data Science.
- Знайти оптимальні метрики аудіосигналу для аналізу. Побудувати декілька наборів даних для навчання і тестування моделей класифікації.
- Розробити моделі класифікації з використанням різних алгоритмів та конфігурацій. Порівняти їх та зробити висновки про їх ефективність.
- Розробити бачення модифікованої програми класифікації з використанням всіх минулих напрацювань.

Об'єкт дослідження:

Процеси класифікації аудіофайлів та акустичні властивості сигналів.

Предмет дослідження:

Інформаційні засоби, методи, моделі управління даними і процесами при аналізі акустичних властивостей сигналів і класифікації аудіофайлів.

Методи дослідження:

- Аналіз теоретичних засад обробки аудіосигналів.
- Порівняння та вибір метрик аудіосигналу.
- Веб-скрапинг.
- Оптимізація розмірності даних.
- Методи нормалізації даних.
- Теоретичний аналіз методів класифікації в машинному навчанні.
- Співставлення результатів моделей класифікації.
- Засоби мови програмування Python, її стандартної бібліотеки, статистичного фреймворку Pandas, засобів веб-скрапингу, бібліотек вилучення акустичних характеристик librosa та машинного навчання sklearn.

Наукова новизна одержаних результатів полягає у:

- Розробці набору метрик для класифікації музики.
- Створенні набору алгоритмів для жанрової класифікації пісень.
- Порівнянні ефективності побудованих моделей та рекомендацій щодо практичного застосування напрацювань.

Теоретична значимість дослідження полягає у проведеному аналізі та відбору оптимальних чисельних метрик обробки аудіосигналів.

Практична значимість роботи полягає у виявленні оптимальної архітектури, стеку технологій та дизайнерських рішень для розробки системи аналізу музичної індустрії.

Структура роботи:

Робота складається зі вступу, трьох розділів, що діляться на підрозділи, висновків, списку використаних джерел з 54 найменувань. Загальний обсяг роботи складає 118 сторінок.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ЗАСАДИ ТА ПІДХОДИ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ АНАЛІЗУ МУЗИЧНОЇ ІНДУСТРІЇ

1.1 Цифрове представлення звуку

Одним з найпростіших, але найбільш цілісним сучасним форматом аудіофайлів є .wav. Звукові хвилі у ньому оцифровуються шляхом вибірки з дискретних інтервалів, відомих як частота дискретизації (як правило, 44,1 кГц для аудіо з CD-якістю — це 44 100 семплів в секунду) [4].

Кожен семпл є амплітудою хвилі в певному часовому інтервалі, де глибина в бітах (або динамічний діапазон сигналу) визначає, наскільки деталізованим буде семпл (зазвичай 16 біт, тобто семпл може варіюватися від 65 536 значень амплітуди).

В обробці сигналів семплінг — це перетворення безперервного сигналу в серію дискретних значень. Частота дискретизації — це кількість семплів за певний фіксований проміжок часу. Висока частота дискретизації призводить до меншої втрати інформації, але до великих обчислювальних витрат.

1.2. Сутність та задачі аудіоаналізу

Аудіоаналіз полягає у вилученні корисної інформації зі звукових сигналів для подальшого її аналізу, класифікації, зберігання, пошуку, синтезу тощо. У звуковому аналізі використовується два ключових різновиди методів збору та обробки даних: перший вид полягає у дослідженні специфіки сприйняття звуку людиною, другий — досліджує суто фізичні властивості звуку, такі як амплітуда, спотворення, частотна характеристика. Зазвичай аналіз першого виду, тобто дослідження інформації, як вона сприймається людиною, виконується після другого, який надає для нього конкретні чисельні характеристики [3][6].

Фізичні властивості аудіосигналу можна аналізувати багатьма різними способами, залежно від того, яку інформацію потрібно отримати від сигналу. Підходи до аналізу сигналів включають:

- Аналіз амплітуди (гучності та гейну);
- Аналіз частотної області;
- Амплітудно-частотну характеристику;
- Коефіцієнт нелінійних спотворень;
- Аналіз фази коливань;
- Перехресні перешкоди;
- Інтермодуляційні спотворення [2].

Програмні аудіоаналізатори, що в нинішній час замінюють апаратні, регулярно використовуються на різних галузях музичного виробництва, таких як обробка в режимі реального часу, змішування та мастеринг. Ці продукти, як правило, використовують алгоритм швидкого перетворення Фур'є (ШПФ), щоб забезпечити візуальне представлення аналізованого сигналу. Типи відображуваної інформації включають частотний спектр, стереополе, об'ємне поле, спектрограму тощо [3][5].

1.3. Інструменти аудіоаналізу в Python

Pyo — це модуль Python, написаний на C для створення скриптів цифрової обробки сигналів. Він містить класи для різних типів обробки аудіосигналів, які користувач може використовувати послідовно для обробки сигналів безпосередньо в скрипті або проекті Python, або маніпулювати ними в режимі реального часу за допомогою інтерпретатора [7].

Інструменти в модулі pyo пропонують примітиви, такі як математичні операції над звуковим сигналом, попередня обробка сигналу (фільтри, затримки, генератори синтезу тощо), а також складні алгоритми для створення грануляції звуку та інших творчих звукові маніпуляцій.

pyAudioAnalysis — це відкрита бібліотека Python, яка надає широкий спектр функціональних можливостей, пов’язаних зі звуком, зосереджуючись на вилученні об’єктів, класифікації, сегментації та візуалізації. За допомогою цієї бібліотеки можна вилучати звукові властивості та подання, класифікувати невідомі звуки, застосовувати зменшення розмірності для візуалізації звукових даних та уніфікації вмісту, виконувати контрольовану та неконтрольовану сегментацію, виявляти звукові події та виключати періоди тиші з тривалих записів та ін. [8]

Dejavu Project — це проект звукових «відбитків» із відкритим вихідним кодом на Python. Він може запам’ятовувати записаний звук, прослухавши його один раз і взявши його відбиток. Потім, відтворюючи пісню та записуючи вхід мікрофона, або зчитуючи файл з диску, Dejavu намагається зіставити звук із «відбитками пальців», що містяться в базі даних, повертаючи пісню або відтворений запис. Dejavu якісно розпізнає точний сигнал при помірній кількості шуму [9].

Mingus — це пакет для Python, який використовується програмістами, музикантами, композиторами та дослідниками для створення та дослідження музики. Це просунутий крос-платформний пакет теорії музики та нотації для Python з підтримкою файлів MIDI та функцією відтворення файлу [10].

З його допомогою можна працювати з теорією музики, створювати редактори, навчальні інструменти та інші програми, які потребують обробки або відтворення музики. В основі Mingus лежить теорія музики, яка включає такі поняття, як інтервали, акорди, гами та прогресії.

hYPerSonic — це фреймворк Python/C для побудови та маніпулювання комунікаційними лініями обробки звуку, призначений для управління в режимі реального часу. Це — робота на низькому рівні, де кожен байт має значення, тому він включає об'єкти для генераторів, фільтрів, файлів-іо, звукових карт та операцій з пам'яттю. На даний момент бібліотека працює на Linux та OSX [11].

Pydub — це простий та легкий інтерфейс високого рівня, заснований на ffmpeg і під впливом jQuery. Він маніпулює звуком, додаючи ефекти, теги id3, нарізуючи фрагменти, об'єднуючи звукові доріжки [12].

Loris — це програмний пакет для моделювання та обробки звуку з відкритим вихідним кодом, заснований на Reassigned Bandwidth-Enhanced Additive Sound Model [14]. Він підтримує модифікований ресинтез та маніпуляції з даними моделі, такі як модифікація шкали часу та частоти та морфінг звуку. Незважаючи на те, що це бібліотека C ++, внутрішній інтерфейс Loris підтримує мову програмування Python, а також пропонуються файли інтерфейсу SWIG, щоб API можна було легко розширити на безліч інших мов [13].

Librosa — це модуль Python для аналізу звукових сигналів в цілому, але переважно спрямований на музику. Він включає все необхідне для побудови системи MIR (вилучення музичної інформації), зокрема, інструменти маніпуляції над файлом та функції вилучення властивостей [16].

В рамках дослідження для обробки аудіофайлів умісним буде обрати пакет Librosa, через сумісність з засобами Data Science в Python та бібліотекою Scikit Learn, а також вичерпність і простоту інтерфейсу. Додатково для обробки метаданих файлів буде використано Tinytag.

1.4. Музичні властивості в librosa

Кожен звуковий сигнал містить численні властивості. Однак ми повинні виділити характеристики, які мають відношення до проблеми, яку ми намагаємось вирішити. Процес вилучення властивостей для їх використання для аналізу називається feature extraction [16][28].

Chroma feature або **chromagram —** це метрика музики, яка полягає у визначенні нот, що складають кожен семпл, залежно від його висоти. Функції на основі хромаграми, які також називають "профілями класу висоти", є потужним інструментом для аналізу музики, висоти звуку якої можна суттєво класифікувати (часто за дванадцятьма категоріями) і що використовує

рівномірно-темперований стрій. Однією з головних властивостей хромаграм є те, що вони враховують гармонійні та мелодійні характеристики музики, залишаючись стійкими до змін тембру та інструментарію [17].

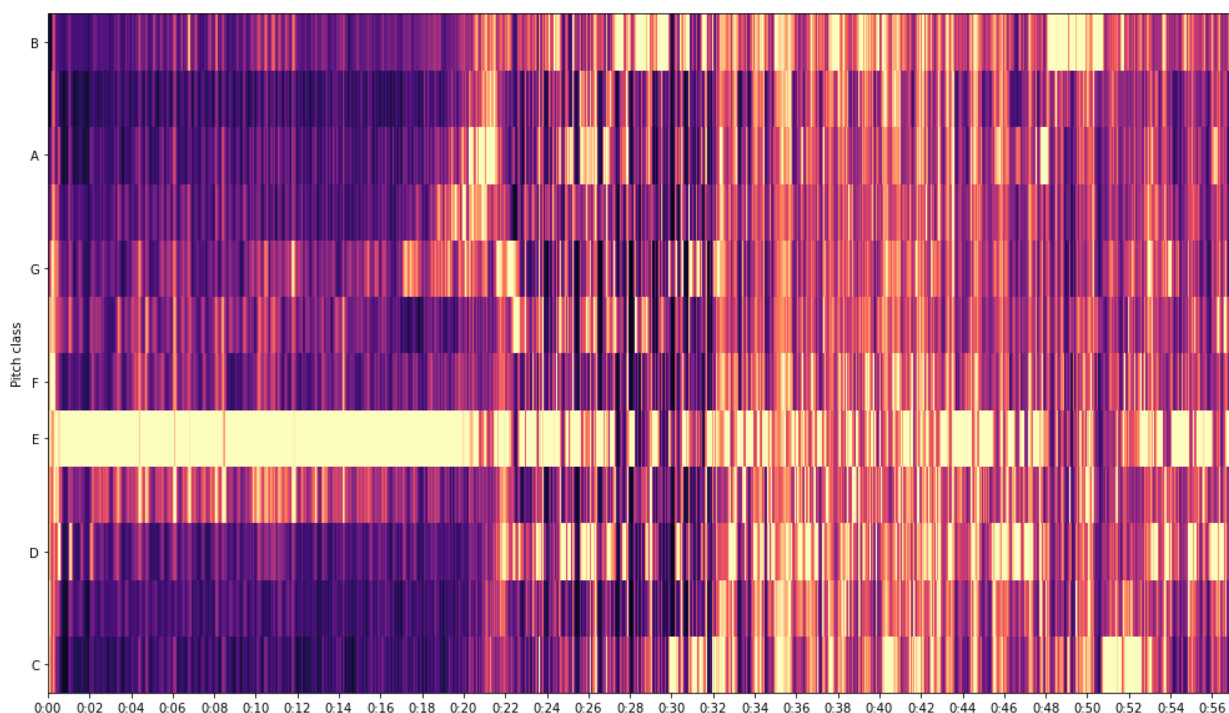


Рисунок 1.1 — Хромаграма пісні Red Hot Chili Peppers «Can’t Stop»

Кепстральні мел-частоти (MFC) — це представлення короткочасного спектру потужності звуку, засновані на лінійному косинусному перетворенні спектральної густини потужності (функції, що описує розподіл потужності сигналу залежно від частоти, тобто потужність, що припадає на одиничний інтервал частоти) на нелінійній мел-шкалі частоти [18].

Мел — це психофізична одиниця висоти звуку, що враховує специфіку сприйняття звуку людиною, і обраховується за формулою

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right) = 1127 \ln\left(1 + \frac{f}{700}\right) \tag{2.1}$$

де f — висота звуку в Гц.

Кепстр — це один з видів гомоморфної обробки сигналу, функція оберненого перетворення Фур’є від логарифму спектра потужності сигналів. Кепстр можна записати виразженням:

$$C_s(q) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \ln |S(\omega)|^2 e^{i\omega q}\, d\omega \quad (2.2),$$

де S(w) — спектр вхідного сигналу.

Мел-кепстральні коефіцієнти (MFCC) — це коефіцієнти, які в сукупності складають MFC. Вони походять від кепстрального подання аудіокліпу. Різниця між кепструмом та MFCC полягає в тому, що в MFCC смуги частот рівномірно розташовані за мел-шкалою, що наближується до відтворення сприйняття слуховою системою людини більш точно, аніж лінійно рознесені смуги частот, що використовуються в кепструмі. Така трансформація частоти може забезпечити краще збереження музичної інформації, наприклад, при стисненні аудіофайлу [18].

Алгоритм побудови MFCC:

- Виконується перетворення Фур'є сигналу.
- Потужності спектру, отриманого вище, наносяться на мел-шкалу, використовуючи віконну вагову функцію.
- Беруться логарифми потужностей на кожній з мел-частот.
- Береться дискретне косинусне перетворення списку потужностей отриманих значень як сигналу.
- MFCC — це амплітуди результуючого спектру [19].

Цей процес може мати варіації, наприклад, відмінності у формі або інтервалі вікон, що використовуються для масштабування шкали.

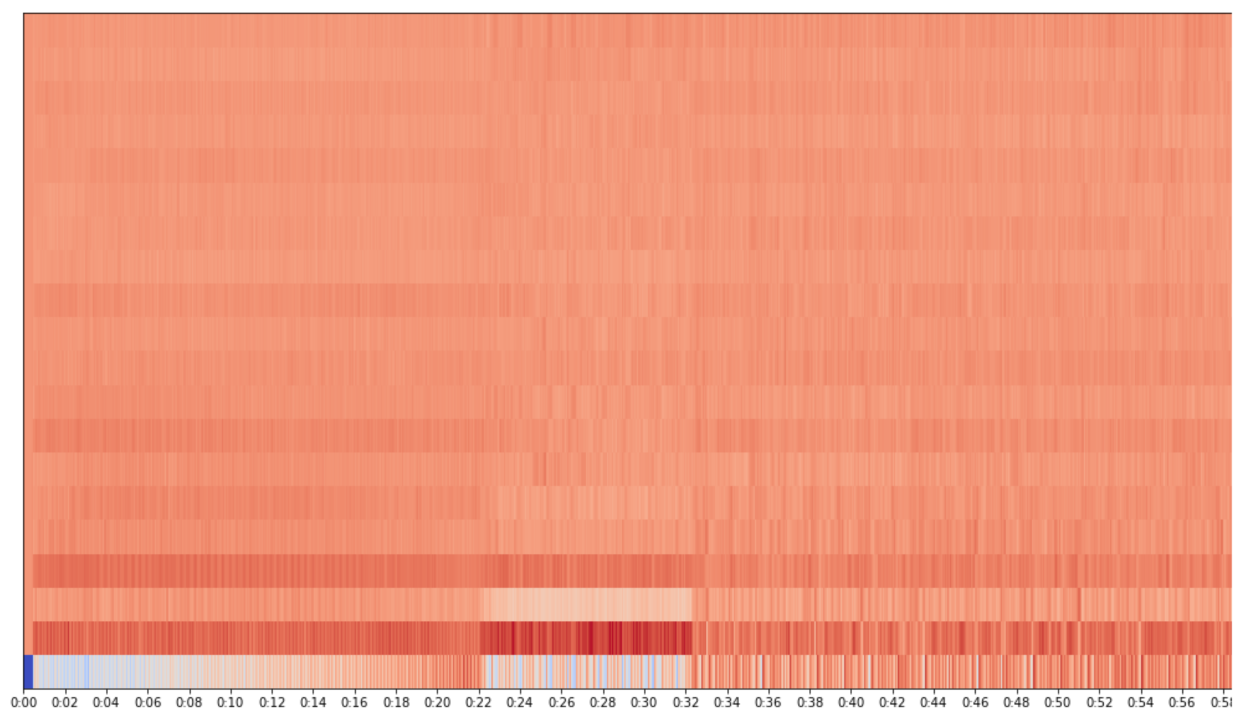


Рисунок 1.2 — Мел-кепстральні коефіцієнти пісні Red Hot Chili Peppers «Can’t Stop»

Спектральний контраст:

Кожен кадр спектрограми S розділюється на діапазони. Для кожного діапазону енергетичний контраст оцінюється шляхом порівняння середньої енергії у верхньому квантилі (пікова енергія) з енергією нижнього квантилю (енергія долини). Значення високої контрастності, як правило, відповідають чітким вузькосмуговим сигналам, тоді як значення низької контрастності відповідають широкосмуговим шумам [20].

Спектральний контраст представляє силу спектральних піків та провалів у кожному діапазоні окремо, щоб відображати відносні спектральні характеристики.

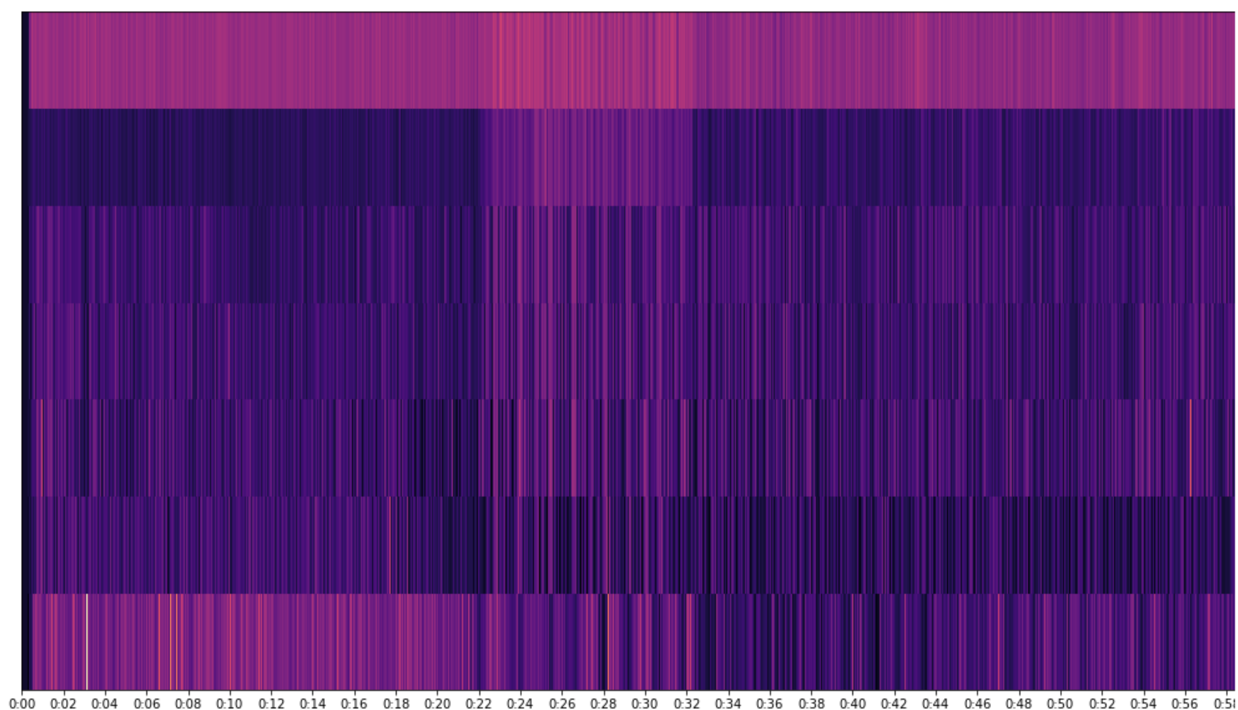


Рисунок 1.3 — Спектральний контраст пісні Red Hot Chili Peppers «Can't Stop»

Tonnetz — обчислення тональних центроїдів. Надає проекцію хроматичних властивостей на 6-мірну шкалу, відображуючи чисту квінту, велику та малу терцію у вигляді двох координат.

Ця модель відображає 12-мірні вектори хромаграми у внутрішній простір 6-D багатогранника; класи висоти відображаються на вершинах цього багатогранника. Близькі гармонічні відносини, такі як квінти та терції, виглядають як малі евклідові відстані. Щоб розробити міру гармонічних змін для кадру n, обчислюється евклідівська відстань між кадрами аналізу n + 1 та n − 1. Пік у функції виявлення означає перехід від однієї гармонічно стабільної області до іншої [21].

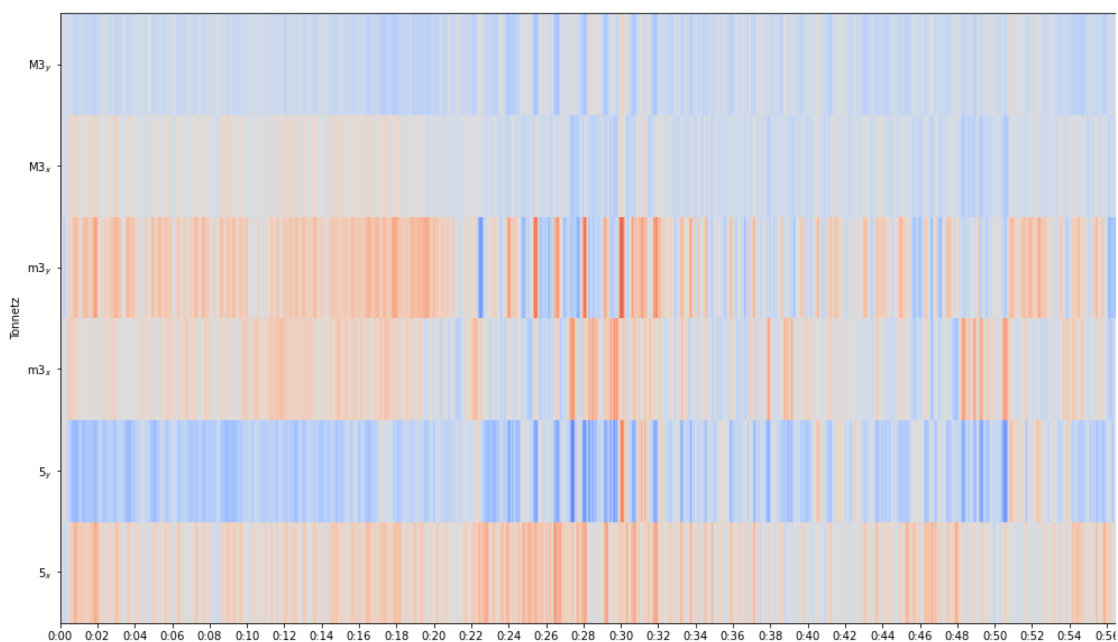


Рисунок 1.4 — Tonnetz пісні Red Hot Chili Peppers «Can’t Stop»

Спектральний коефіцієнт площинності або тональності, також відомий як ентропія Вінера — це міра, що використовується в цифровій обробці сигналу для характеристики звукового спектру. Спектральна площинність, як правило, вимірюється в децибелах і забезпечує спосіб кількісно визначити, наскільки тоноподібний звук, на відміну від шумоподібного [22].

Тон в цьому контексті полягає у великій кількості піків, або резонансній структурі у спектрі потужності, на відміну від плоского спектра, що характерний для білого шуму. Висока спектральна площинність (наближається до 1,0 для білого шуму) вказує на те, що спектр має однакову кількість потужності у всіх спектральних смугах — це буде звучати подібно до білого шуму, і графік спектру буде здаватися відносно рівним. Низька спектральна площинність (наближається до 0,0 для чистого тону) вказує на те, що спектральна сила зосереджена у відносно невеликій кількості смуг — це, як правило, звучить як суміш синусоїд, а спектр виглядає "пишним".

Спектральна площинність обчислюється діленням середнього геометричного спектру потужності на середнє арифметичне спектру потужності.

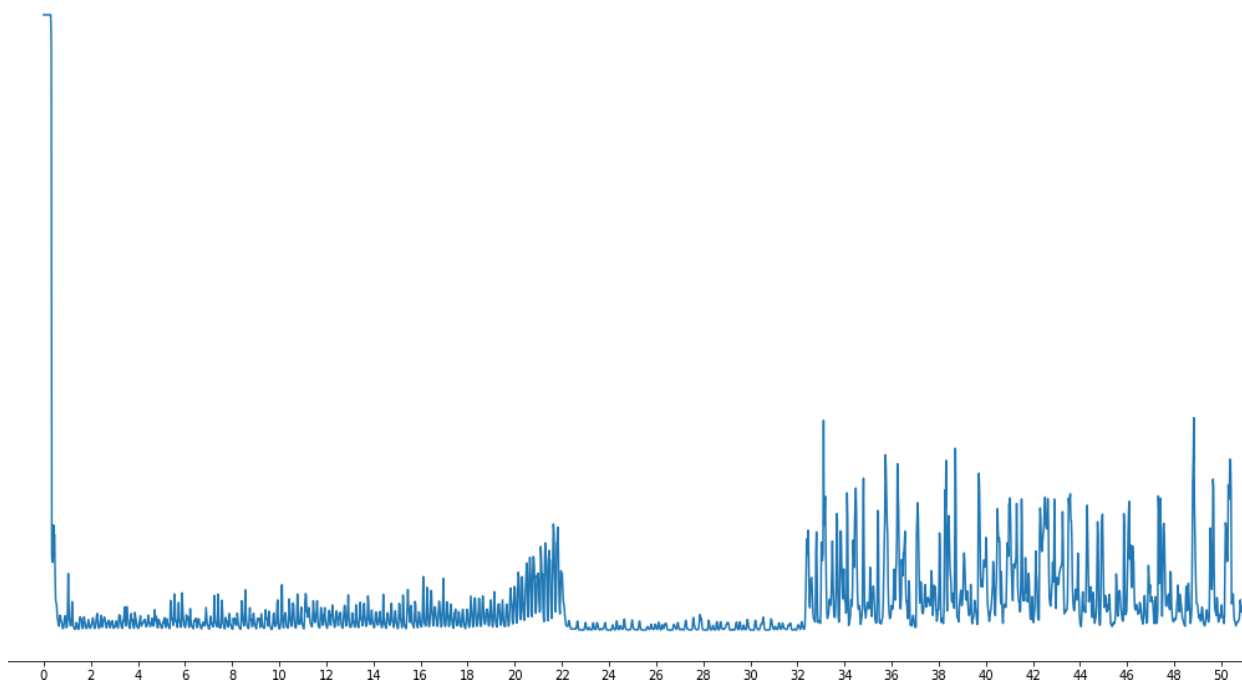


Рисунок 1.5 — Спектральна площинність пісні Red Hot Chili Peppers «Can't Stop»

Частота згортання визначається для кожного кадру як така точка на спектрограмі, що щонайменше заданий процент (0,85 за замовчуванням) енергії спектра в цьому кадрі міститься в цій точці та нижче. Це можна використовувати, наприклад, для наближення до максимальної або мінімальної частоти, встановивши процент на значення, близьке до 1 або 0.

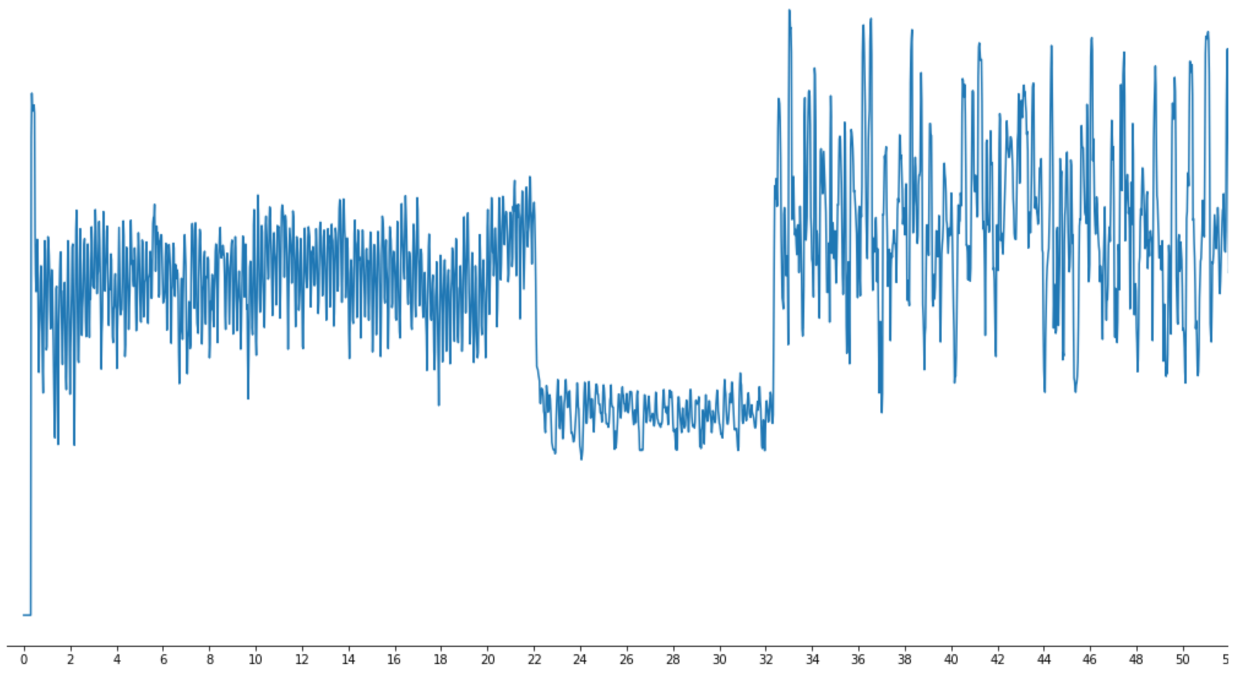


Рисунок 1.6 — Частота згортання пісні Red Hot Chili Peppers «Can’t Stop»

РОЗДІЛ 2.
ЧИСЕЛЬНЕ ПРЕДСТАВЛЕННЯ АУДІОСИГНАЛІВ. ЗБІР, ОПТИМІЗАЦІЯ ТА НОРМАЛІЗАЦІЯ ДАНИХ

2.1. Опис підходу до представлення аудіофайлу як сирих даних

Для побудови датасетів створюємо окремий Python скрипт (load.py). В ньому спершу імпортуємо всі необхідні залежності:

- Модуль os: для пошуку і завантаження аудіофайлів.
- NumPy: для роботи з масивами і пошуку статистик по вибіркам.
- Librosa: основна бібліотека для обробки аудіофайлу і вилучення властивостей [23].
- Sklearn: для нормалізації даних для майбутнього використання в моделях машиного навчання [25].
- Pandas: для створення, обробки і збереження датасетів.
- Tinytag: для вилучення метаданих з файлів.

Функція read_directory виконує рекурсивний обхід директорії, збираючи всі аудіофайли, записує результати обробки в датафрейм та зберігає його.

```
def read_directory(dirname, df_file_name, intermediate_save=False):
    analysis_df = DataFrame()

    for (address, dirs, files) in os.walk(os.path.join(AUDIO_DIR, dirname)):
        for file in files:
            analysis_df = process_song(os.path.join(address, file), analysis_df, save=intermediate_save, filename=df_file_name)

    if not intermediate_save:
        analysis_df.to_csv(os.path.join(PROJECT_DIR, df_file_name))
```

Рисунок 2.1 — Функція read_directory

Функція process_song обробляє задану пісню. Спершу файл librosa та метадані tinytag зчитуються з диску.

```python
def process_song(path, analysis_df=None, save=True, title=None, artist=None, filename=None):
    if analysis_df is None:
        analysis_df = DataFrame()

    # song = load(os.path.join(AUDIO_DIR, '01 Can\'t Stop.mp3'), sr=SR)[0]
    song = load(path, sr=SR)[0]
    metadata = TinyTag.get(path)
```

Рисунок 2.2 — Функція process_song

Далі зберігаємо назву, виконавця (якщо не знайдені — встановлюємо з файлової системи). Розраховуємо BPM: для цього в librosa існує функція beat_track. Тривалість пісні отримуємо з функції get_duration. Для нормалізації даних тривалість берем відносно максимально допустимої (константа); нормалізацію BPM виконує окрема функція.

```python
# General song stuff
if metadata.title:
    title = metadata.title
elif not title:
    title = os.path.split(path)[1]
if metadata.artist:
    artist = metadata.artist
elif not artist:
    artist = os.path.basename(os.path.dirname(path))
# song_harmonic, song_percussive = hpss(song)
bpm = beat_track(song, sr=SR)[0]
bpm = normalize_bpm(bpm)
duration = get_duration(song) / MAX_SONG_DURATION
if duration > 1:
    duration = 1
```

Рисунок 2.3 — загальні характеристики пісні

Оскільки BPM може бути будь-яким числом, і вільно ділитись або множитись на 2, призведемо його до стандартизованого вигляду від 50 до 100 за допомогою рекурсивної функції, а потім візьмемо частку від ділення на 100.

```python
def normalize_bpm(bpm):
    if bpm > 100:
        bpm = normalize_bpm(bpm/2)
    elif bpm < 50:
        bpm = normalize_bpm(bpm*2)
    else:
        bpm /= 100
    return bpm
```

Рисунок 2.4 — функція нормалізації BPM

Викликаємо функції з модулю librosa.features з параметрами, що визначені константами. Для результатів функцій, що повертають значення поза межами [0; 1], додатково викликаємо функцію нормалізації minmax_scale з пакету sklearn (тут названа normalize).

```python
song_measures = {
    'chroma_cqt': chroma_cqt(song, sr=SR, hop_length=MEASURE_HOP_LENGTH),
    'mfcc': normalize(mfcc(song, sr=SR), axis=0),
    'spectral_contrast': normalize(spectral_contrast(song, sr=SR, hop_length=MEASURE_HOP_LENGTH), axis=0),
    'tonnetz': normalize(tonnetz(song, sr=SR), axis=0),
    # 'zero_crossing_rate': zero_crossing_rate(song, hop_length=MEASURE_HOP_LENGTH),
    'spectral_rolloff': normalize(spectral_rolloff(song, sr=SR, hop_length=MEASURE_HOP_LENGTH)),
    'spectral_flatness': normalize(spectral_flatness(song, hop_length=MEASURE_HOP_LENGTH)),
}
```

Рисунок 2.5 — Вилучення властивостей пісні

Параметри за замовченням:

- FRAME_LENGTH — довжина фрейму, на які поділяється кожна пісня;
- SR — роздільність аудіофайлу;
- MEASURE_HOP_LENGTH — довжина кластеру для функцій вилучення властивостей;
- MAX_SONG_DURATION — максимально допустима тривалість в секундах.
- PROJECT_DIR, AUDIO_DIR — для пошуку і збереження файлів.

Оскільки об’єм даних всієї пісні сягає величезного розміру, статистика по всій пісні буде малоінформативною. Тому пісню, а, точніше, її вилучені

властивості, доцільно поділити на фрагменти, або, в глосарії librosa, фрейми. Для цього використовується функція frame (тут — перейменована на frame_func) [24]. Для коректного опрацювання в циклі вісі результатів поділу (багатовимірного масиву) необхідно переставити, щоб поділ на фрейми йшов першим, за допомогою функції np.moveaxis.

```
frame_length = int(FRAME_LENGTH / MEASURE_HOP_LENGTH)  # valid for all
for key, value in song_measures.items():
    # if len(value.shape) > 1 and value.shape[0] == 1:
    #     value = np.reshape(value, value.shape[1])
    # value = np.clip(value, 0, 1)
    value = frame_func(value, frame_length=frame_length, hop_length=frame_length, axis=-1)
    song_measures[key] = np.moveaxis(value, -1, 0)
```

Рисунок 2.6 — Поділ характеристик на фрейми

Далі оброблюємо фрейми в циклі, додаючи кожен з них як рядок в датафрейм. Для кожного фрейму вказуємо параметри всієї пісні, а також, важливо — його позицію. Вилучені властивості вносимо до датафрейму за допомогою окремої функції через велику кількість колонок в них [47].

```
frame_df_idx = analysis_df.shape[0] if analysis_df.shape else 0
frames = frame_func(song, frame_length=FRAME_LENGTH, hop_length=FRAME_LENGTH, axis=0)
frames_count = len(frames)
for frame_idx in range(frames_count):
    analysis_df.at[frame_df_idx, 'title'] = title
    analysis_df.at[frame_df_idx, 'artist'] = artist
    analysis_df.at[frame_df_idx, 'bpm'] = bpm
    analysis_df.at[frame_df_idx, 'duration'] = duration

    frame_idx_rel = frame_idx / frames_count
    analysis_df.at[frame_df_idx, 'frame position'] = frame_idx_rel

    # frame = frames[frame_idx]

    for key, value in song_measures.items():
        add_array_stats_to_df(value[frame_idx], key, analysis_df, frame_df_idx)

    frame_df_idx += 1
```

Рисунок 2.7 — Запис характеристик пісні в датафрейм

Для властивостей, що зберігаються у вигляді двовимірного масиву, функція add_array_stats_to_df викликається рекурсивно для кожного рядку. В іншому випадку вона розраховує наступні статистики:

- Мінімум;
- Максимум;
- Медіана;
- Мода;
- Середнє значення;
- Середньоквадратичне відхилення;
- Коефіцієнт варіації;

та записує їх в датафрейм, кожну в окрему колонку. Функція np.average дозволяє знайти зважене середнє, але тут воно здається недоцільним, а без параметрів вона дублює np.mean, тому вона була пропущена [46].

```
def add_array_stats_to_df(array, colname, df, index):
    if len(array.shape) == 1:
        statistics = {
            'min': np.amin(array),
            'max': np.amax(array),
            'median': np.median(array),
            # 'average': np.average(array),
            'mean': np.mean(array),
            'std': np.std(array),
            'var': np.var(array),
        }
        for key, value in statistics.items():
            df.at[index, f'{colname} {key}'] = value
    else:
        for i in range(len(array)):
            add_array_stats_to_df(array[i], f'{colname} {i}', df, index)
```

Рисунок 2.8 — Розрахунок статистик властивостей

Таким чином, наприклад, для MFCC, що містить 20 колонок для кожного фрейму, в кінцевому датасеті створюється 120 колонок, а загальна їх кількість

складає 287. Тривалість фрейму складає приблизно 5 секунд, тому кожна пісня в середньому нараховує 30-100 фреймів (і, відповідно, рядків в датасеті).

Фрагмент датасету supervised_train.csv, що містить 90 пісень груп Led Zeppelin та Red Hot Chili Peppers, показано в таблиці 2.1.

Таблиця 2.1

	title	**artist**	**bpm**	**duration**	**frame position**	**chroma_cqt 0 min**	**chroma_cqt 0 max**	**chroma_cqt 0 median**
0	Achilles Last Stand	Led Zeppelin	0.717773	0.348778	0.000000	0.124634	0.821098	0.345807
1	Achilles Last Stand	Led Zeppelin	0.717773	0.348778	0.009524	0.058713	0.574840	0.246888
2	Achilles Last Stand	Led Zeppelin	0.717773	0.348778	0.019048	0.083662	0.739834	0.255896
3	Achilles Last Stand	Led Zeppelin	0.717773	0.348778	0.028571	0.093190	0.829006	0.324271
4	Achilles Last Stand	Led Zeppelin	0.717773	0.348778	0.038095	0.124313	0.695623	0.365715

2.2 Зменшення розмірності даних для коректного порівняння моделей

В створеному датасеті пісні розділені на фрейми, і кожний фрейм — окремий рядок. Це може бути корисним для точної класифікації конкретної пісні за сукупним/осереднім/модальним результатом декількох фреймів. Але для цілей даної роботи це не становить перевагу, оскільки робить такий датасет трьовимірним. Намір ж роботи полягає у порівнянні моделей, побудованих на цьому датасеті з аналогічними на іншому, двовимірному датасеті жанрової класифікації Spotify. Таким чином, для нормалізації цього датасету необхідно перетворити його з двовимірного на тривимірний.

Так зване зниження розмірності в області машинного навчання полягає у використанні певного методу зіставлення точок даних у вихідному багатовимірному просторі з низькорозмірним простором. Суть зменшення розмірності полягає в тому, щоб вивчити функцію відображення f(x) = y, де x — це вираз вихідної точки даних (в більшості випадків використовується форма векторного виразу), а y — це низькорозмірний векторний вираз після відображення точок даних. Зменшення розмірності може бути поділено на *відбір ознак* та *виділення ознак* [49].

Метод відбору ознак намагається знайти підмножину вихідних змінних, які називаються ознаками чи атрибутами. Є три стратегії — стратегія фільтрування (наприклад, накопичення ознак), стратегія обгортання (наприклад, пошук відповідно до точності) і стратегія вкладення (вибираються ознаки для додавання або видалення в міру побудови моделі, що базується на помилках прогнозування). У деяких випадках аналіз даних, такий як регресія або класифікація, може бути здійснений у редукованому просторі більш точно, ніж у вихідному просторі.

Проекція ознак перетворює дані із простору високої розмірності до простору малої розмірності. Перетворення даних може бути лінійним, як у методі основних компонент (МГК), але також є велика кількість технік нелінійного зниження розмірності. Для багатовимірних даних можна використовувати тензорне уявлення зниження розмірності через полілінійне навчання підпросторів.

Для наборів даних високої розмірності (тобто з числом розмірностей більше 10) зниження розмірності зазвичай здійснюється перед застосуванням методу k-найближчих сусідів з метою уникнути ефекту «прокляття розмірності». Виділення ознак і зниження розмірності можуть бути скомбіновані в один крок за допомогою методу головних компонентів (МГК), лінійного дискримінантного аналізу (ЛДА), канонічного кореляційного аналізу (ККА) або невід'ємного розкладання матриці (НМР) як попередній крок з наступним угрупуванням за допомогою K-NN на векторі ознак у

просторі редукованої розмірності. У машинному навчанні цей процес називається також малорозмірним вкладенням. Для будь-яких наборів даних високої розмірності (наприклад, коли здійснюється пошук подібності у відеопотоці, даних ДНК, або у часовому ряді високої розмірності) використання швидкого наближеного K-NN пошуку за допомогою методів locality sensitive hashing, випадкової проекції або інших високорозмірних технік пошуку схожості з арсеналу надвеликих баз даних може виявитися єдиним можливим варіантом [50].

Для даної задачі ж першочергова необхідність полягає не просто в оптимізації кількості ознак, а саме зменшення розмірності датасету з трьовимірної матриці на двовимірну. Існуючі методи зменшення розмірності не вирішують цю задачу, тому доведеться застосувати власний метод.

Сутність його полягатиме в зміні підходу до фреймування файлу з поділу на фрагменти фіксованої довжини на поділ на фіксовану кількість фреймів, довжина яких визначається відносно довжини поділюваного файлу. Тоді, оскільки тепер кількість фреймів є однаковою для кожного запису в датасеті, то інформацію відносно кожного фрейму можливо буде винести в колонки. Таким чином, кількість рядків датасету зменшиться, а кількість колонок зросте, але кількість інформації фактично залишиться незмінною.

Внесемо бажані правки до скрипту load.py. Передусім необхідно встановити константну кількість фреймів.

```
FRAME_LENGTH = 16
SR = 22050
MEASURE_HOP_LENGTH = 512
MAX_SONG_DURATION = 1800
PROJECT_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
AUDIO_DIR = os.path.join(PROJECT_DIR, 'audio_data')
```

Рисунок 2.9 — Фіксована кількість фреймів

Функція обчислення метрик тепер використовуватиме повний масив замість його фрагменту. Функція заповнення датасету побудована для обробки

багатовимірних масивів шляхом рекурсивних викликів, але для правильної обробки необхідно встановити коректний порядок осей в масиві, тому його теж потрібно оновити. Правильний порядок осей було знайдено експериментально.

```
86    for key, value in song_measures.items():
87        # if len(value.shape) > 1 and value.shape[0] == 1:
88        #     value = np.reshape(value, value.shape[1])
89        # value = np.clip(value, 0, 1)
90        value = frame_func(value, frame_length=FRAME_LENGTH, hop_length=FRAME_LENGTH, axis=-1)
91        song_measures[key] = np.moveaxis(value, 1, 0)
```

Рисунок 2.10 — Виклик функції фреймування з новою змінною кількості та новий порядок осей

Тепер, коли кожній пісні відповідатиме один рядок датасету, вкладений цикл заповнення спільних ознак по всій пісні можна прибрати. Для простої ідентифікації варто також додати назву файлу на диску.

```
93    analysis_df_idx = analysis_df.shape[0] if analysis_df.shape else 0
94    analysis_df.at[analysis_df_idx, 'filename'] = filename
95    analysis_df.at[analysis_df_idx, 'title'] = title
96    analysis_df.at[analysis_df_idx, 'artist'] = artist
97    analysis_df.at[analysis_df_idx, 'bpm'] = bpm
98    analysis_df.at[analysis_df_idx, 'duration'] = duration
```

Рисунок 2.11 — Заповнення метаданих файлу

Надалі необхідно переробити виклик функції заповнення датасету даними аналізу з по-фреймового виклику на виклик для всього файлу одразу. Тепер немає необхідності слідкувати за позицією фрейму:

```
frame_idx_rel = frame_idx / frames_count
analysis_df.at[frame_df_idx, 'frame position'] = frame_idx_rel

# frame = frames[frame_idx]

for key, value in song_measures.items():
    add_array_stats_to_df(value[frame_idx], key, analysis_df, frame_df_idx)

frame_df_idx += 1
```

Рисунок 2.12 — Попередня версія введення результатів аналізу

Але замість одновимірного масиву в функцію вводиться двовимірний:

```
100     # frame_idx_rel = frame_idx / FRAME_LENGTH
101     # analysis_df.at[frame_df_idx, 'frame position'] = frame_idx_rel
102
103     for key, value in song_measures.items():
104         add_array_stats_to_df(value, key, analysis_df, analysis_df_idx)
105
```

Рисунок 2.13 — Нова, спрощена версія введення результатів

Скрипт створення датасету також було несуттєво модифіковано з урахуванням нової ієрархії файлів в директорії.

Порівняємо зміну виду датасету в Jupyter. Минулий формат даних:

```
In [4]: test_df = os.path.join('..', 'data', 'test.csv')
        test_df = pd.read_csv(test_df)
        test_df
```

											...			
11	11	Can't Stop	Red Hot Chili Peppers	0.922852	0.148941	0.6875	0.211227	0.471077	0.272542	0.302030	...	0.538798	0.549044	0.
12	12	Can't Stop	Red Hot Chili Peppers	0.922852	0.148941	0.7500	0.169893	0.438924	0.296627	0.306623	...	0.575956	0.570628	0.
13	13	Can't Stop	Red Hot Chili Peppers	0.922852	0.148941	0.8125	0.140502	0.360132	0.210541	0.216668	...	0.539891	0.545423	0.
14	14	Can't Stop	Red Hot Chili Peppers	0.922852	0.148941	0.8750	0.199647	0.328591	0.286208	0.273906	...	0.558470	0.554508	0.
15	15	Can't Stop	Red Hot Chili Peppers	0.922852	0.148941	0.9375	0.200181	0.339850	0.284133	0.277622	...	0.619672	0.616052	0.

16 rows × 288 columns

Рисунок 2.14 — Попередня версія датасету аналізу. Була отримана на проміжковій версії скрипту, тому містить 16 рядків (за числом фреймів)

```
In [11]: list(test_df.columns)

Out[11]: ['Unnamed: 0',
          'title',
          'artist',
          'bpm',
          'duration',
          'frame position',
          'chroma_cqt 0 min',
          'chroma_cqt 0 max',
          'chroma_cqt 0 median',
          'chroma_cqt 0 mean',
          'chroma_cqt 0 std',
          'chroma_cqt 0 var',
          'chroma_cqt 1 min',
          'chroma_cqt 1 max',
          'chroma_cqt 1 median',
          'chroma_cqt 1 mean',
          'chroma_cqt 1 std',
          'chroma_cqt 1 var',
          'chroma_cqt 2 min',
```

Рисунок 2.15 — Назви колонок минулого датасету. Число вказує на колонку в матриці метрики, здебільшого від 0 до 7 або 12 за числом нот

В попередньому датасеті число колонок складає 288. 6 з них — метадані пісні, решта — результати виміру. Новий формат датасету:

```
In [13]: test_df = os.path.join('..', 'data', 'test2.csv')
         test_df = pd.read_csv(test_df)
         test_df
```

Out[13]:

	Unnamed: 0	title	artist	bpm	duration	chroma_cqt 0 0 min	chroma_cqt 0 0 max	chroma_cqt 0 0 median	chroma_cqt 0 0 mean	chroma_cqt 0 0 std	...	spectral_flatness 14 0 median	spectral_flatness 14 0 mean	spe
0	0	Can't Stop	Red Hot Chili Peppers	0.922852	0.148941	0.04383	1.0	0.498108	0.537726	0.271999	...	0.029353	0.052512	

1 rows × 4517 columns

Рисунок 2.16 — Новий вид датасету. Одній пісні відповідає один рядок

```
In [14]: list(test_df.columns)

Out[14]: ['Unnamed: 0',
          'title',
          'artist',
          'bpm',
          'duration',
          'chroma_cqt 0 0 min',
          'chroma_cqt 0 0 max',
          'chroma_cqt 0 0 median',
          'chroma_cqt 0 0 mean',
          'chroma_cqt 0 0 std',
          'chroma_cqt 0 0 var',
          'chroma_cqt 0 1 min',
          'chroma_cqt 0 1 max',
          'chroma_cqt 0 1 median',
          'chroma_cqt 0 1 mean',
          'chroma_cqt 0 1 std',
          'chroma_cqt 0 1 var',
          'chroma_cqt 0 2 min',
          'chroma_cqt 0 2 max',
```

Рисунок 2.17 — Назви колонок нового датасету

Кількість колонок нового датасету складає 4517, з яких 5 — метадані. 4512 колонок вимірювань — це рівно в 16 раз більше аніж 282 в попередньому, тобто вимірювання мультиплікуються кількістю фреймів, як і очікується. В назві колонки виміру перше число — індекс фрейму від 0 до 15, друге — індекс рядку в матриці виміру.

2.3 Збір даних для навчання

Для навчання моделей необхідно зібрати архів пісень з заздалегідь відомою класифікацією по жанрам, потім цей архів необхідно буде обробити розробленим скриптом та отримати тренувальний датасет.

Для того, щоб хоч певною мірою вилучити користь з такої великої кількості ознак, як чотири з половиною тисячі, передбачається обробити приблизно 500 пісень, поділених на 5-8 жанрів. Насправді, і цієї кількості буде замало для того, щоб повністю проявити потенціал методу, але, як виявилось в ході виконання роботи, така підготовка даних являє суттєву обчислювальну

складність, і навіть названа кількість вимагає близько доби неперервної обробки файлів.

Далі: існує чимало принципових способів визначити музичний жанр конкретного запису, від веб-скрапінгу до використання заздалегідь сформованої бібліотеки. Для даної задачі видається достатнім датасет класифікації Spotify, вивантажений з сайту Kaggle в процесі розробки мети роботи. Цей датасет містить велику кількість пісень з бібліотеки Spotify, їх жанрову класифікацію, музичні характеристики, вилучені сервісом, та метадані.

В якості альтернативи одному датасету можливо було б використати автоматизований веб-пошук по декількох сайтах з парсингом результатів. Втім, для теми роботи це становить надмірну складність, оскільки обраний датасет надає цілком вичерпну інформацію.

Окрім метаданих пісень, необхідний також безпосередній доступ до файлів задля їх обробки бібліотекою librosa. Звісно, самих файлів або їх вмісту датасет не містить, тому необхідно буде завантажити їх окремо.

Спершу необхідно дізнатись, які саме пісні шукати. З корисних в цьому плані метаданих датасет містить назву пісні й автора, а також ID файлу в Spotify. Останній нам не знадобиться, оскільки Spotify є пропрієтарним сервісом, і вивантаження великої кількості пісень вимагатиме затрат на закупку ліцензій.

В Інтернеті можна знайти велику кількість спеціалізованих безкоштовних хостингів аудіофайлів. Однак жоден з них не являє собою вичерпну бібліотеку, яка міститиме все, чого можна зажадати, а більша частина навпаки суттєво обмежена (здебільшого, тим, що завантажили самі користувачі). Таким чином, опора на такі сервіси буде означати або необхідність відфільтрувати пісні за їх наявністю в конкретному сховищі, або необхідність створити декілька парсерів для різних сайтів, що може викликати велику кількість проблем у зв’язку з різним характером можливих помилок.

Крім того, існує додаткова проблема різних форматів, які, можливо, необхідно буде переводити до єдиного.

Нестандартне ж рішення проблеми збору музичних файлів — використання такого не спеціалізованого, але великого та вичерпного відеохостингу, як YouTube. Він має підрозділ для музики, яка технічно являє собою відеофайли; втім, колекція контенту на цьому сервісі точно задовільнить будь-які потреби [38].

Перша проблема полягає в тому, що YouTube використовує власний програвач відео, відмінний від простого браузерного. Цей плеєр не дозволяє завантажити файл безпосередньо, і задачу вивантаження файлів звично вирішують за допомогою сторонніх сервісів [39][40].

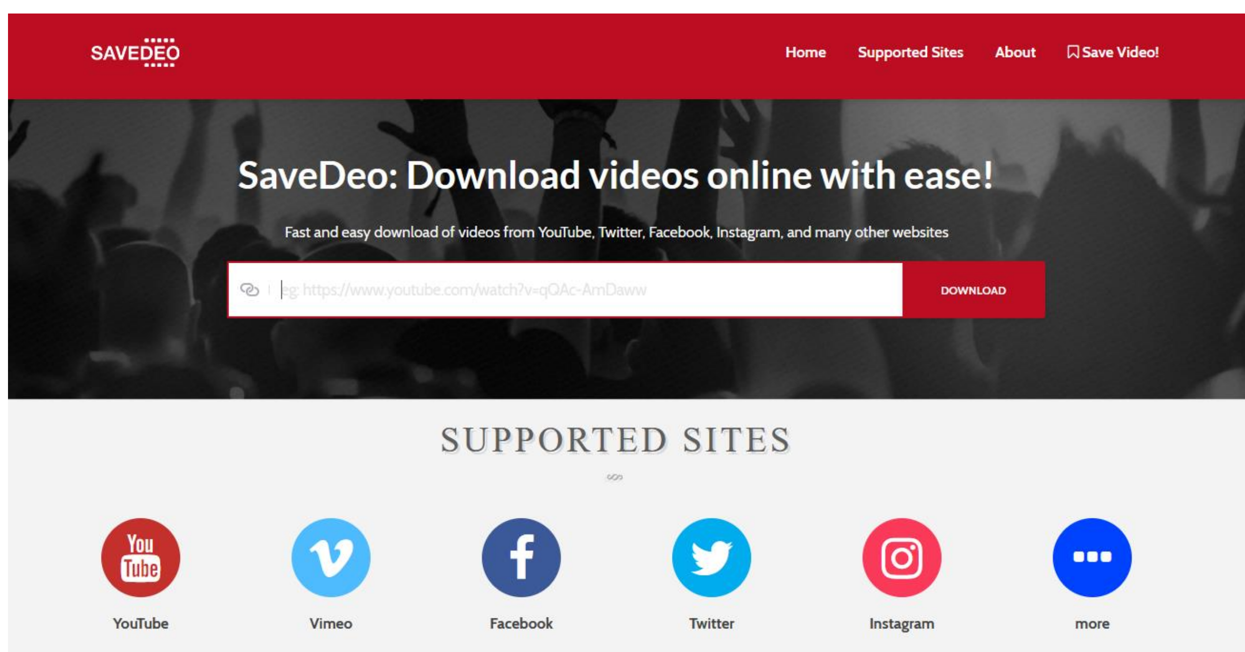


Рисунок 2.18 — Головна сторінка сервісу SaveDeo

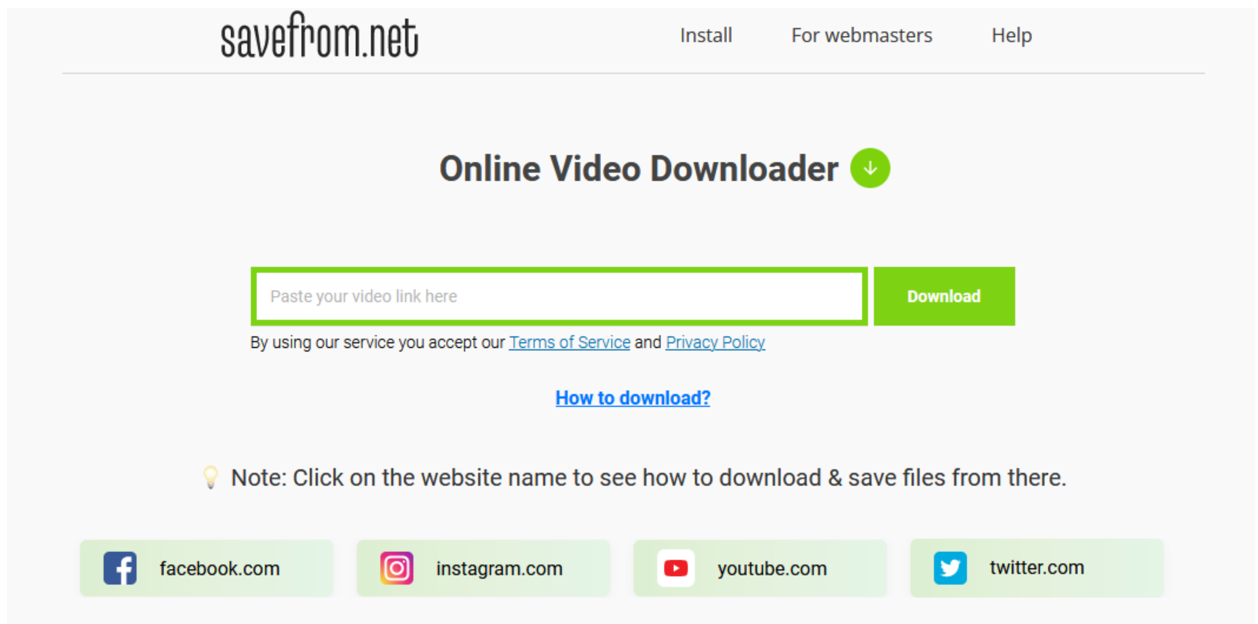


Рисунок 2.19 — Головна сторінка сервісу Savefrom.net

Втім, використання сторонніх сервісів викликає складність у вигляді можливих обмежень їх API або обсягу скачуваних даних: наприклад, доступ розробника може продаватись за окремим тарифом. До того ж скачування через проксі іншого серверу буде повільнішим.

В якості альтернативи можна звернутися до API самого YouTube.

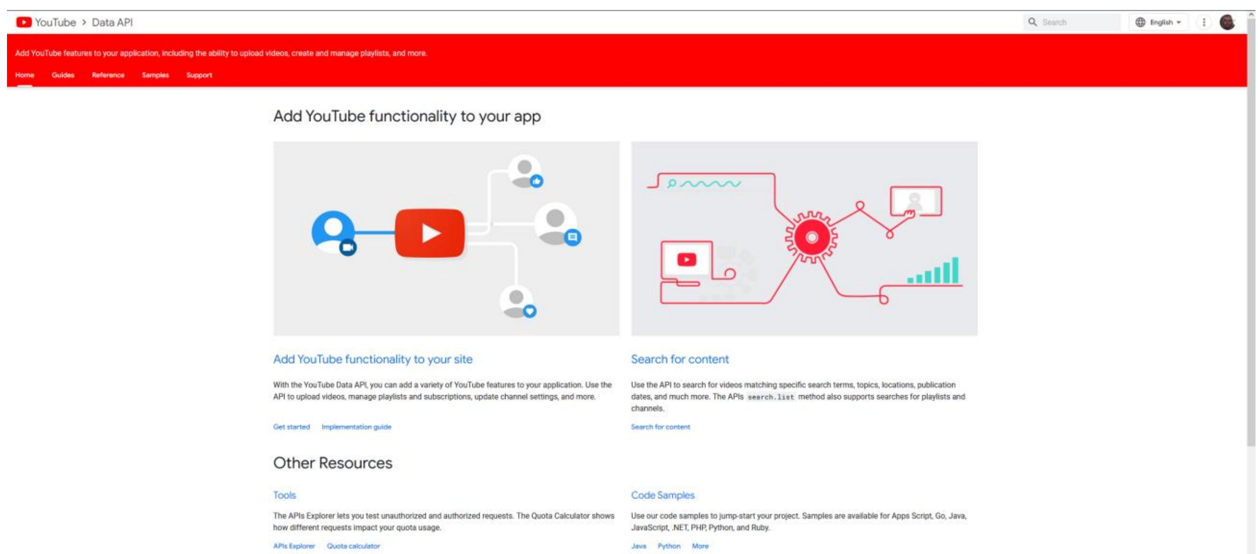


Рисунок 2.20 — Головна сторінка документації API YouTube

API YouTube також накладає певні вимоги та обмеження для використання [41]. Передусім, потрібен обліковий запис Google, щоб отримати доступ до Google API Console, запитати ключ API та зареєструвати свій застосунок. Далі необхідно створити проект на Google Developers Console та отримати облікові дані авторизації, щоб програма могла надсилати запити API. Після створення проекту потрібно впевнитись, що YouTube data API є однією зі служб, для використання якої зареєстрована ця програма:

1) Перейти на консоль API і знайти проект, який щойно було зареєструвано;
2) Відвідати сторінку ввімкнених API.
3) У списку API переконатися, що встановлено статус ON для YouTube Data API v3 [36].

Окрім того, YouTube data API використовує квоту, щоб гарантувати, що розробники використовують службу за призначенням і не створюють програми, які несправедливо знижують якість послуг або обмежують доступ для інших. Усі запити API, включаючи недійсні запити, оплачуються щонайменше в один бал. Знайти квоту, доступну для вашої програми, можна на консолі API. Проекти, які підтримують API даних YouTube, мають рівень квоти за замовчуванням у **10 000 умовної одиниць на день**, що вважається достатнім для більшості користувачів API. Квота за замовчуванням, яка може бути змінена, допомагає YouTube оптимізувати розподіл квоти та масштабувати інфраструктуру таким чином, щоб це було більш значущим для користувачів API. Побачити поточний рівень використання квоти можна на сторінці Квоти в Консолі API.

Google обчислює використання квоти, призначаючи вартість кожному запиту. Різні види операцій мають різну вартість квот. Наприклад:

- Операція читання, яка отримує список ресурсів – каналів, відео, списків відтворення – зазвичай коштує 1 одиницю.

- Операція запису, яка створює, оновлює або видаляє ресурс, зазвичай коштує 50 одиниць.
- **Пошуковий запит коштує 100 одиниць.**
- Вартість завантаження відео становить 1600 одиниць.

До того ж варто звернути увагу на наступні два моменти, оскільки обидва вони впливають на використання квоти: по перше, якщо програма викликає метод, наприклад, search.list, який повертає кілька сторінок результатів, кожен запит на отримання додаткової сторінки результатів збільшує витрату квоти. По друге, методи YouTube Live Streaming API технічно є частиною API даних YouTube, і виклики цих методів також потребують квоти. Таким чином, методи API для прямої трансляції також наведено в таблиці Quota Calculator.

Таблиця "Вартість квоти для запитів API" показує вартість квоти для кожного методу API. Зважаючи на ці правила, можливо оцінити кількість запитів, які програма може надіслати за день, не перевищуючи квоту.

Quota costs		
resource	**method**	**cost**
activities	list	1
captions	list	50
	insert	400
	update	450
	delete	50
channelBanners	insert	50
channels	list	1
	update	50
channelSections	list	1
	insert	50
	update	50
	delete	50
comments	list	1
	insert	50
	update	50
	markAsSpam	50
	setModerationStatus	50
	delete	50

Рисунок 2.21 — Фрагмент таблиці Quota Costs

Втім, вищеозначене не допомогає вирішити дві ключові проблеми поставленої задачі: завантаження відео, а перед тим — пошук за назвою. YouTube API підтримує лише аналіз метаданих відео та пов'язаного контенту, такого як плейлисти, канали та коментарі, але сам вміст відео завантажити не дозволяє. Друга проблема — пошук відео [37]. Використання пошуку в API коштує 100 балів API із 10 000, що надаються на добу; таким чином, завантаження 500 пісень займе тиждень очікування [42].

На щастя, для безпосереднього скачування відео за ID існують додаткові сервіси, деякі з яких було згадано вище. Однак для розробки скрипту бажано знайти сервіс, спрямований не тільки на користувачів, а й для девелоперів. На щастя, існує програма, що задовольняє цій вимозі. Це — мінімалістична програма **youtube_dl**, розроблена на мові Python [43].

youtube-dl — це програма командного рядка для завантаження відео з YouTube.com та кількох інших сайтів. Для її запуску потрібен інтерпретатор Python версії 2.6, 2.7 або 3.2+, і вона є крос-платформленою. Програма працює на пристроях Unix, в Windows або macOS. Вона опублікована у суспільне надбання, що означає, що будь-який розробник можете змінювати її, розповсюджувати або використовувати як завгодно [44].

youtube-dl дозволяє завантажити відео за ідентифікатором, але ідентифікатор ще необхідно отримати. Пошук на YouTube застосувати не вийде через обмеження квоти; постає альтернативна ідея: використати пошуковий сервіс. Google може мати або не мати власний API для виконання пошуку, але найшвидший і найпростіший результат — спробувати скопіювати запит із браузера.

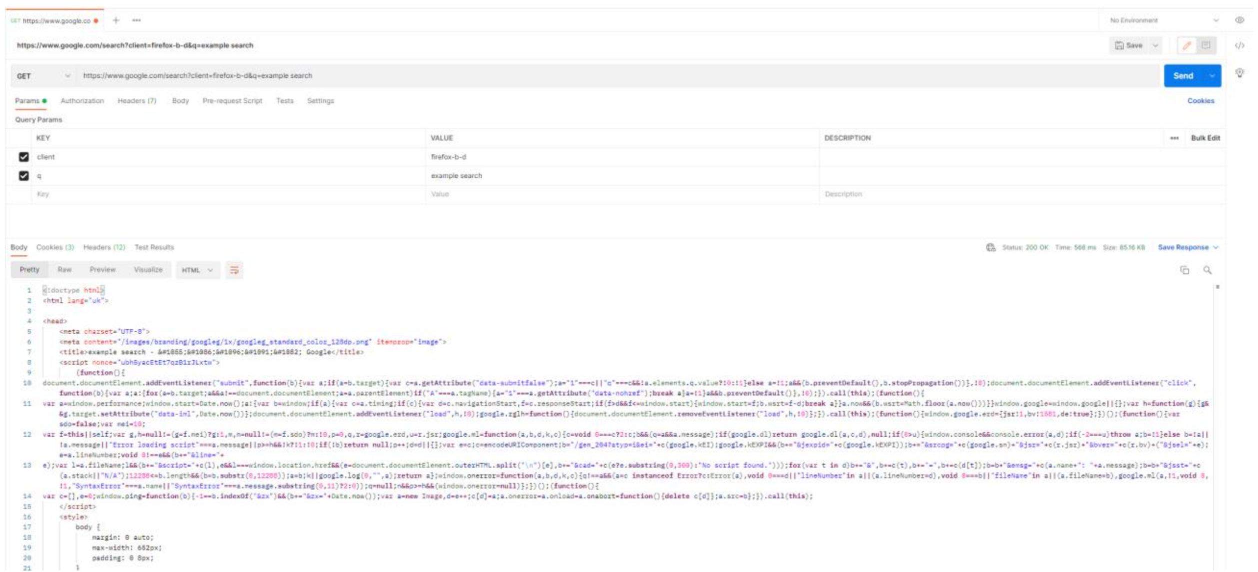


Рисунок 2.22 — Приклад запиту, скопійованого з браузеру

Цих технологій має бути достатньо для виконання задачі. Можна приступати до розробки нового скрипту.

Передусім необхідно становити пакет youtube-dl (запит будемо робити стандартною бібліотекою, встановлювати нічого не потрібно). Можна використати команду pip install youtube-dl, або додати пакет потрібної версії до файлу вимог та виконати pip install -r requirements.txt.

```
1  mutagen==1.45.1
2  numpy==1.20.3
3  pandas==1.2.4
4  librosa==0.8.1
5  scikit-learn==0.24.2
6  tinytag==1.8.1
7  notebook==6.4.11
8  youtube-dl==2021.12.17
```

Рисунок 2.23 — Вміст файлу requirements.txt

Передусім необхідно імпортувати залежності. Бібліотека BeautifulSoup корисна для парсингу HTML відповіді на пошуковий запит [51]. Бібліотека traceback використовується для позначення помилки в процесі виконання головної функції youtube_dl. Бібліотека requests необхідна для формування

HTTP запитів на пошук. Бібліотека os використовується для операцій над файлами на диску.

```
from bs4 import BeautifulSoup
import traceback
import requests
import os

import pandas as pd
import youtube_dl

from metrics import AUDIO_DIR, PROJECT_DIR
```

Рисунок 2.24 — Імпорт залежностей

Зовнішні бібліотеки — pandas для зчитування датасету (щоб дізнатись назви бажаних пісень) та, власне, youtube_dl.

З попереднього скрипту імпортуємо шляхи до директорій, щоб не ініціалізувати зайві змінні.

Далі йдуть константи. Перший блок — query параметри реквесту, безпосередньо скопійовані з браузеру. При чому власне пошукова строка буде додаватись на ходу у два інші параметри, оскільки залежить від рядку зчитуваного датасету.

```
# Update this with params from browser if outdated
GOOGLE_REQUEST_PARAMS = {
    'client': 'firefox-b-d',
    'sxsrf': 'ALiCzsYeQOLBixMg7V9skdV78HeZ9ewXdg:1652270559592',
    'ei': '36V7Yt3aI7GorgTIq76oAw',
    'ved': '0ahUKEwidxpu-s9f3AhUxlIsKHciVDzUQ4dUDCA0',
    'gs_lcp': 'Cgdnd3Mtd2l6EAwyBwgAELADEB5KBAhBGAFKBAhGGABQAFgAYP30CGgBcAB4AIABAIgBAJIBAJgBAMgBAcABAQ',
    'sclient': 'gws-wiz',
}
```

Рисунок 2.25 — Параметри HTTP запиту в Google на пошук

Другий набір констант — параметри для виклику youtube_dl [45]. Бібліотека буде викликатись і як клас Python, і з командної строки завдяки

модулю os. Для цього необхідно зберегти опції в двох різних форматах, один з яких має бути словником, другий — строкою.

```
YOUTUBE_DL_OPTIONS = {
    'format': 'bestaudio/best',
    'postprocessors': [{
        'key': 'FFmpegExtractAudio',
        'preferredcodec': 'mp3',
        'preferredquality': '192',
    }],
    'outtmpl': os.path.join(AUDIO_DIR, '%(title)s.%(ext)s'),
    'cachedir': False,
}
YOUTUBE_DL_CLI_OPTIONS = '--rm-cache-dir --extract-audio --prefer-ffmpeg --audio-format mp3 --prefer-ffmpeg'
```

Рисунок 2.26 — Параметри запуску youtube_dl

Показані параметри позначають завантаження аудіо найкращої можливої якості з використанням конкретного постпроцесору та збереженням у форматі mp3. Крім того, вказується шлях збереження завантажуваного файлу (правда, він буде перезаписаний номером конкретного рядку), а також вказівкою пропускати кешування, яке становить проблеми при завантаженні, а в інакшому випадку кеш регулярно має бути очищений.

Цілком аналогічні параметри, але з відмінним синтаксисом, вказані в строці параметрів для виклику з командної строки.

Будь-яку програму зазвичай доцільно розробляти як набір атомізованих компонент, поділених на чітку структуру, наприклад, класами або функціями. Це допомагає в масштабуванні ПЗ та відслідкуванні помилок. Тут варто дотримуватись такого ж підходу, тому розділимо окремі логічні компоненти на різні функції.

Перша з них буде віднаходити ідентифікатор конкретного відео за назвою пісні, використовуючи пошук в Google. Функція використовує визначений вище словник параметрів запиту. Константний словник метаданих реквесту оновлюється двома змінними, що несуть пошукову строку, до якої додається фільтр за веб-сайтом.

```
def find_video_ids(title):
    title = title.replace('\'', '').replace('"', '')
    params = GOOGLE_REQUEST_PARAMS
    params.update({
        'q': f'{title} site:youtube.com',
        'oq': f'{title} site:youtube.com',
    })
    response = requests.request('GET', 'https://www.google.com/search', params=GOOGLE_REQUEST_PARAMS)
    page = BeautifulSoup(response.text, 'html.parser')
    video_ids = []
    for search_result in page.find_all('div', {'class': 'egMi0 kCrYT'}):
        redirect_url = search_result.find('a').attrs['href']
        try:
            redirect_url = redirect_url.split('watch%3Fv%3D')[1].split('&sa=U')[0]
            video_ids.append(redirect_url)
        except IndexError:
            pass
    return video_ids
```

Рисунок 2.27 — Вміст функції пошуку ідентифікаторів

Далі виконується GET запит за допомогою стандартної бібліотеки на URL, що використовується для пошуку. Запит також включає вищеозначені параметри URL.

Результат приходить в форматі HTML. З нього необхідно вилучити бажані посилання на відео, оскільки вони містять в собі ідентифікатори. Для цього використовується бібліотека BeautifulSoup. З відповіді HTTP запиту, що зберігається в текстовому форматі, формується об’єкт, який в термінології бібліотеки називається «суп». В цьому «супі» можна ітерувати по нодам, що задовільняють вказаним HTML селекторам.

Ітерування здійснюється по нодам, які являють собою результати пошуку, відображувані на одній сторінці. В кожній ноді нескладно вилучити бажаний елемент <a> з атрибутом href, який і містить посилання на відео.

Посилання містить в собі ідентифікатор, який нескладно вилучити за допомогою функції поділу строки. Вилучений ідентифікатор можна додати до результуючого масиву.

Втім, варто зауважити, що фільтр по сайту YouTube.com не гарантує, що кожен результат буде саме посиланням на відео: результатом пошуку можуть

бути послання на інший контент, такий як канали або плейлисти. Проблема в тому, що такий URL матиме структуру, що ламатиме парсер строки.

Однак для нас будь-які посилання, окрім відео, не становлять інтересу, тому немає сенсу розробляти окремий парсер на кожен вид контенту. Достатньо буде створити обгортку з конструкції try … except для ігнорування будь-яких URL відмінного формату.

Залишається заповнити масив результатів та повернути його після проходження циклу.

Тепер необхідно використати знайдений ідентифікатор для завантаження файлу на диск. Для цього розробимо окрему функцію.

```
def download_youtube_audio(video_ids, filename=None):
    ydl_opts = YOUTUBE_DL_OPTIONS.copy()
    if filename is not None:
        ydl_opts.update({
            'outtmpl': os.path.join(AUDIO_DIR, f'{filename}.%(ext)s')
        })
    with youtube_dl.YoutubeDL(ydl_opts) as ydl:
        views = {}
        for video_id in video_ids:
            try:
                metadata = ydl.extract_info(youtube_link(video_id), download=False)
                views[video_id] = metadata['view_count']
                if len(views) >= 3:
                    break
            except youtube_dl.utils.DownloadError:
                continue
        # video_id = sorted(views.keys(), key=lambda x: views[x], reverse=True)[0]
        # ydl.download([youtube_link(video_id)])

    video_id = sorted(views.keys(), key=lambda x: views[x], reverse=True)[0]
    os.chdir(PROJECT_DIR)
    os.system(f'youtube-dl {YOUTUBE_DL_CLI_OPTIONS} -o {ydl_opts["outtmpl"]} {youtube_link(video_id)}')
```

Рисунок 2.28 — Функція завантаження файлу

youtube_dl дозволяє встановити шаблонізований шлях до цільового файлу в параметрі outtmpl. Назва файлу має містити індекс рядка датасету для легшої ідентифікації, тому оновлюємо значення цього параметра бажаною назвою, яку вимагаємо в параметрі filename функції. Стандартний словник налаштувань доцільно скопіювати перед оновленням на випадок, якщо деякі зміни в ньому становитимуть шкоду для інших файлів.

Далі в обгортці with ініціюється головний клас youtube_dl. Така обгортка дозволяє автоматично використати механізм класу для закриття файлів та об'єктів пам'яті, будь-яких інших дій після завершення роботи з ним. Після цього можна обробити кожний ідентифікатор в циклі.

Файли бажано сортувати за кількістю переглядів, щоб скачати найпопулярніший. На щастя, youtube_dl надає функціонал завантаження метаданих відео без його вмісту. Для цього використовується метод extract_info з параметром download, встановленим на False. Обов'язковим інпутом youtube_dl є повне посилання на відео. Сформувати його нам допоможе окрема невелика функція.

```
def youtube_link(video_id):
    return f'https://www.youtube.com/watch?v={video_id}'
```

Рисунок 2.29 — Генерація прямого посилання на відео

Використання такої функції замість посилання, вилученого зі сторінки результатів, дозволяє отримати посилання в стандартизованому вигляді, з правильним протоколом, без префіксів та query параметрів.

Вивантажені метадані надходять в форматі масиву. Вилучаємо кількість переглядів з цього масиву та зберігаємо її разом з ідентифікатором відео. Видається достатнім зібрати лише декілька перших результатів (інші можуть бути більш переглюваними, але не відповідати бажаному запиту), і завершити цикл, якщо їх число перевищує цю кількість, зокрема 3.

Певні відео бібліотека вивантажити нездатна: наприклад, відео, до яких закритий доступ з даного регіону. Для таких випадків варто огорнути фрагмент пошуку метаданих блоком try … except, що перехоплює вказаний клас помилок та пропускає даний ідентифікатор.

В результаті циклу виявляється зібрано словник із трьох ідентифікаторів з різною кількістю переглядів. Тепер необхідно обрати найбільш переглюваний ID, для чого масив ключів словника можна сортувати в

оберненому порядку. Ключ сортування — строкова функція, що повертає кількість переглядів для даного ключа. У відсортованому масиві беремо перший елемент, це ідентифікатор найпопулярнішого відео серед перших трьох.

Взагалі, для завантаження відео можна використати метод того самого класу, що використовується для огляду метаданих. Але експериментальне застосування показало, що в бібліотеці наявні проблеми з очищенням кешу, яке в деяких випадках не відбувається, незважаючи на прямий виклик відповідного методу. Виклик ж виконуваного файлу бібліотеки через консоль такі проблеми здебільшого вирішує.

Для правильної інтерпретації відносного шляху до файлу необхідно змінити робочу директорію за допомогою методу os.chdir(). Після цього можна виконати консольну команду «повторного» запуску бібліотеки, для чого використовується метод system того ж модулю. Повний текст команди, який має містити значення декількох змінних, допомагає сформувати F-строка. Команда включає назву виконуваного файлу, набір опцій, визначений в константах, персоналізований шлях до цільового файлу та посилання на відео. Команда може бути виконана в тому ж віртуальному середовищі, в якому запускається скрипт, але для її спрацювання необхідно додати шлях до її .exe файлу (у Windows) до системної змінної PATH. Команда завантажить файл до вказаної директорії, встановить вказану назву.

Тепер, коли функції пошуку назви та завантаження пісні розроблена, необхідно розробити невелику функцію, що викликатиме обидві з них для кожної пісні у зчитуваному нею датасеті.

```python
def download_songs_from_dataset(path, offset=0):
    target_df = pd.read_csv(path, index_col=0)
    target_df = target_df.head(550)
    for row_idx in range(offset, len(target_df)):
        row = target_df.iloc[row_idx]
        index, artist, title = row.name, row['artist_name'], row['track_name']
        print(f'Song # {row_idx}: {index} {artist}: {title}')
        try:
            download_youtube_audio(find_video_ids(f'{artist} {title}'), filename=index)
        except Exception as e:
            # Skip any file with error
            print(e)
            traceback.print_exc()
```

Рисунок 2.30 — Функція обробки пісень з датасету

Функція зчитує датасет за вказаним шляхом, обирає з нього перші декілька пісень. Спершу було вирішено зібрати 512 пісень (але в подальшому їх кількість незначно виросла), тому для урахування можливих помилок або недоступних файлів обираються перші 550 рядків.

Ітерація відбувається з певного індексу, 0 за замовченням. Це зроблено для цілей дебагу, оскільки помилки в завантаженні окремих пісень вимагали допрацювання скрипту. Після виправлення помилки скрипт перезапускався, і, щоб уникнути повторного скачування файлів, попередньо встановлювався офсет.

Функція вилучує назву пісні та виконавця з датасету, виводить їх в консоль для відстеження прогресу, та надсилає у попередньо визначені функції. При цьому результат функції збору ідентифікаторів безпосередньо надається у функцію завантаження. Цільова назва при завантаженні дорівнює індексу строки датасету.

Незважаючи на досить ретельну роботу над обробкою різних варіантів посилань та вмісту, залишається велика кількість потенційних проблем, і вирішувати кожну з них заради одного відео недоцільно. Для масового ігнорування будь-яких «проблемних» відео використовується try … except. Помилка даного рядку виводиться у консоль, а цикл продовжує виконання.

Оскільки файли названі за індексами рядків, вилучити незнайдені пісні в подальшому не складатиме проблеми. Для цього в скрипті метрик зберігається також назва файлу. Загалом було пропущено близько 20 відео.

```
if __name__ == '__main__':
    # download_youtube_audio(find_video_ids('Natalie Grant In Christ Alone')[:3])
    download_songs_from_dataset(os.path.join(PROJECT_DIR, 'data', 'sample_df.csv'), 100)
```

Рисунок 2.31 — Виклик загальної функції зі шляхом до датасету зі списком пісень та офсетом 100

Залишається запустити скрипт та дозволити йому завантажити вказану кількість файлів.

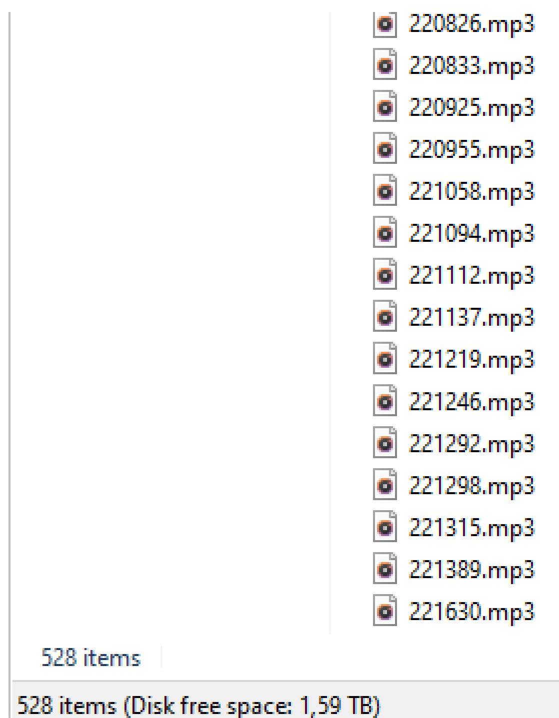


Рисунок 2.32 — Результат роботи скрипту — ~530 завантажених файлів

Після цього запускаємо попередній скрипт з пошуком по цій директорії та формуємо датасет метрик аудіофайлів.

```python
my_metric_df = pd.read_csv(my_metric_df_path, index_col=0)
my_metric_df['index'] = my_metric_df['filename'].apply(lambda x: x.split('.')[0])
my_metric_df = my_metric_df.drop(['title', 'artist', 'filename'], axis=1)
my_metric_df = my_metric_df.set_index('index')
my_metric_df
```

index	bpm	duration	chroma_cqt 0 0 min	chroma_cqt 0 0 max	chroma_cqt 0 0 median	chroma_cqt 0 0 mean	chroma_cqt 0 0 std	chroma_cqt 0 0 var	chroma_cqt 0 1 min	chroma_cqt 0 1 max	...	spectral_flatness 14 0 median	spe
107817	0.759995	0.120496	0.034730	1.0	0.580889	0.554494	0.294984	0.087015	0.029309	1.0	...	0.007559	
107820	0.861328	0.116044	0.033829	1.0	0.556860	0.532828	0.244733	0.059894	0.060821	1.0	...	0.002343	
107905	0.759995	0.100768	0.021219	1.0	0.266448	0.324134	0.225566	0.050880	0.020424	1.0	...	0.010158	
107974	0.993840	0.100484	0.013705	1.0	0.294330	0.320845	0.205709	0.042316	0.008610	1.0	...	0.010849	
108014	0.645996	0.133386	0.000000	1.0	0.152913	0.250648	0.227774	0.051881	0.000000	1.0	...	0.000435	
...	...	...	...	...	...	...	...	...	...	...	...	...	
71180	0.561736	0.173892	0.051963	1.0	0.485455	0.504919	0.171775	0.029507	0.081362	1.0	...	0.010637	
71332	0.645996	0.162449	0.053991	1.0	0.384127	0.441904	0.254483	0.064762	0.062278	1.0	...	0.017099	
71355	0.807495	0.203833	0.106080	1.0	0.698997	0.706382	0.242684	0.058896	0.100176	1.0	...	0.086112	
71472	0.587269	0.174962	0.042289	1.0	0.462695	0.565989	0.334566	0.111934	0.044440	1.0	...	0.038106	
87041	0.833543	0.087010	0.065958	1.0	0.399440	0.467020	0.288387	0.083167	0.038325	1.0	...	0.007939	

523 rows × 4514 columns

Рисунок 2.33 — Отриманий датасет музичних властивостей 523 пісень

2.4 Збір даних для порівняльної моделі. Остаточна нормалізація датасетів

Для оцінки розробленого методу вилучення властивостей передбачається порівняти моделі, побудовані на його основі, з моделями на інших метриках. Для порівняння було знайдено датасет аналізу аудіозаписів сервісом Spotify «SpotifyFeatures.csv» на сайті kaggle.com. Датасет містить назву пісні, виконавця, жанрову класифікацію, ID в сервісі Spotify та метрики аналізу вмісту. В подальшому будемо просто називати цей датасет Spotify.

```
spotify_df = pd.read_csv(spotify_df_path, index_col=0)
spotify_df
```

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	live
114582	Rap	Nipsey Hussle	Racks In The Middle (feat. Roddy Ricch and Hit...	6ya8ejoKgw906Y8LWclqrp	69	0.0825	0.671	233278	0.833	0.000000	F	0.
220748	World	Kick Bong	Flower Power	1ICvWZD5ZuGTsmjhvkDwQV	25	0.2690	0.501	326285	0.672	0.803000	E	0.
218329	World	Henri Texier	Les "Là-bas"	5LawMt3aszO8M3lXzXXesM	40	0.8170	0.830	184427	0.347	0.913000	A	0.
220313	World	Natalie Grant	In Christ Alone	0TrtaB299i1ShhdEtLmxQx	27	0.1390	0.311	332533	0.685	0.000009	C#	0.
52893	Blues	Alice Cooper	Teenage Frankenstein	3YBljZM0WDEQRpVcXyH4vx	28	0.0021	0.589	220667	0.826	0.000009	A	0.
...	...	...	...	...	...	...	...	...	...	...	...	
113664	Pop	Remmy Valenzuela	Mi Princesa	0AxI0XXpjltt7Hi6mzeAhx	68	0.2890	0.679	183600	0.646	0.000000	D#	0.
138177	Reggae	Giant Panda Guerilla Dub Squad	Walk Right Talk Right	1fbDUat77plepvUxbamk4E	21	0.4220	0.799	205833	0.575	0.034100	G	0.
121699	Rap	Freddie Gibbs	Death Row (feat. 03 Greedo)	3YZ5NiutSg7KraH87r4fgm	53	0.0901	0.902	142629	0.660	0.000005	G#	0.
121401	Rap	Kendrick Lamar	untitled 01 | 08.19.2014.	6OkOFxU6j0UufR97VgSnS7	52	0.3470	0.580	247747	0.570	0.000000	D	0.
119909	Rap	Chimbala	Maniqui	1SKZ2kmLyRZUklw9zq9RLL	58	0.3930	0.885	177633	0.805	0.000089	F	0.

54563 rows × 18 columns

Рисунок 2.34 — Фрагмент датасету Spotify

Метадані пісень з датасету Spotify необхідно виокремити, вони будуть використовуватись як референси для класифікації.

```
metadata_df = spotify_df[['genre', 'artist_name', 'track_name']]
display(metadata_df)
metadata_df.to_csv(os.path.join(data_path, 'processed', '512_songs_metadata.csv'))
```

	genre	artist_name	track_name
114582	Rap	Nipsey Hussle	Racks In The Middle (feat. Roddy Ricch and Hit...
220748	World	Kick Bong	Flower Power
218329	World	Henri Texier	Les "Là-bas"
220313	World	Natalie Grant	In Christ Alone
52893	Blues	Alice Cooper	Teenage Frankenstein
...	...	...	...
113664	Pop	Remmy Valenzuela	Mi Princesa
138177	Reggae	Giant Panda Guerilla Dub Squad	Walk Right Talk Right
121699	Rap	Freddie Gibbs	Death Row (feat. 03 Greedo)
121401	Rap	Kendrick Lamar	untitled 01 | 08.19.2014.
119909	Rap	Chimbala	Maniqui

54563 rows × 3 columns

Рисунок 2.35 — Датасет метаданих пісень

Далі необхідно виконати нормалізацію датасету. Чисельні значення в ньому необхідно вирівняти за шкалою від 0 до 1; категоріальні — звести до числових. Так, значення тональності отримує індекс в порядку нот; значення темпу інтерпретується як дроб та зводиться до десятичного вигляду.

```
display(sorted(spotify_metric_df['time_signature'].unique()))
spotify_metric_df['time_signature'] = spotify_metric_df['time_signature'].apply(lambda x: eval(x))
spotify_metric_df[['time_signature']] = scaler.fit_transform(spotify_metric_df[['time_signature']])
spotify_metric_df
```

```
['0/4', '1/4', '3/4', '4/4', '5/4']
```

oularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence
0.69	0.0825	0.671	0.044272	0.833	0.000000	0.727273	0.0726	0.841711	0.0	0.3950	0.224221	0.8	0.702
0.25	0.2690	0.501	0.063720	0.672	0.803000	0.636364	0.1740	0.771605	0.0	0.0607	0.701683	0.6	0.158
0.40	0.8170	0.830	0.034058	0.347	0.913000	0.000000	0.0949	0.660575	1.0	0.0568	0.437590	0.8	0.880
0.27	0.1390	0.311	0.065026	0.685	0.000009	0.363636	0.1740	0.850163	1.0	0.0373	0.480740	0.6	0.144
0.28	0.0021	0.589	0.041635	0.826	0.000009	0.000000	0.0921	0.705130	0.0	0.0344	0.465073	0.8	0.733
...	...	...	...	...	...	...	...	...	...	...	...	...	...
0.68	0.2890	0.679	0.033885	0.646	0.000000	0.545455	0.0874	0.851195	1.0	0.0597	0.355450	0.8	0.694
0.21	0.4220	0.799	0.038534	0.575	0.034100	0.909091	0.0270	0.791178	1.0	0.0469	0.454898	0.8	0.733
0.53	0.0901	0.902	0.025318	0.660	0.000005	1.000000	0.1230	0.827708	1.0	0.3970	0.262889	0.8	0.440
0.52	0.3470	0.580	0.047298	0.570	0.000000	0.454545	0.0897	0.724329	1.0	0.3540	0.332796	0.8	0.143
0.58	0.3930	0.885	0.032637	0.805	0.000089	0.727273	0.0578	0.867084	1.0	0.0541	0.431106	0.8	0.536

× 14 columns

Рисунок 2.36 — Перетворення значення темпу на десятичний дріб.

Після того, як всі ознаки були зведені до чисельного вигляду, вирівнюємо їх за шкалою масштабування від 0 до 1 [48]. Застосовуємо це перетворення як до датасету Spotify, так і до власного датасету (про всяк випадок: в деяких колонках через помилку плаваючої точки з'явилися значення, більші за 1).

```
display(spotify_metric_df)
spotify_metric_df.to_csv(os.path.join(data_path, 'processed', '512_songs_spotify.csv'))
```

	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signat
114582	0.69	0.0825	0.671	0.044272	0.833	0.000000	0.727273	0.0726	0.841711	0.0	0.3950	0.224221	
220748	0.25	0.2690	0.501	0.063720	0.672	0.803000	0.636364	0.1740	0.771605	0.0	0.0607	0.701683	
218329	0.40	0.8170	0.830	0.034058	0.347	0.913000	0.000000	0.0949	0.660575	1.0	0.0568	0.437590	
220313	0.27	0.1390	0.311	0.065026	0.685	0.000009	0.363636	0.1740	0.850163	1.0	0.0373	0.480740	
52893	0.28	0.0021	0.589	0.041635	0.826	0.000009	0.000000	0.0921	0.705130	0.0	0.0344	0.465073	
...	...	...	...	...	...	...	...	...	...	...	...	...	
113664	0.68	0.2890	0.679	0.033885	0.646	0.000000	0.545455	0.0874	0.851195	1.0	0.0597	0.355450	
138177	0.21	0.4220	0.799	0.038534	0.575	0.034100	0.909091	0.0270	0.791178	1.0	0.0469	0.454898	
121699	0.53	0.0901	0.902	0.025318	0.660	0.000005	1.000000	0.1230	0.827708	1.0	0.3970	0.262889	
121401	0.52	0.3470	0.580	0.047298	0.570	0.000000	0.454545	0.0897	0.724329	1.0	0.3540	0.332796	
119909	0.58	0.3930	0.885	0.032637	0.805	0.000089	0.727273	0.0578	0.867084	1.0	0.0541	0.431106	

54563 rows × 14 columns

Рисунок 2.37 — Повністю нормалізований датасет Spotify

Власний датасет містить дані, повторювані для кожного спостереження через специфіку їх збору. Можемо видалити ті колонки, що містять єдине значення у всіх рядках.

```
useless_columns = []
for column in custom_df.columns:
    if len(custom_df[column].unique()) == 1:
        useless_columns.append(column)
custom_df = custom_df.drop(useless_columns, axis=1)
```

Рисунок 2.38 — Функція очистки зайвих ознак

Залишається лише створити функцію семплювання датасетів для вибору тренувальних та тестувальних наборів.

```python
def split_data(length=64):
    """
    return: (train_custom, train_spotify, train_target), (test_custom, test_spotify, test_target)
    """
    training_rows = custom_df.index.to_series().sample(length)
    return (
        custom_df[custom_df.index.isin(training_rows)],
        spotify_df[spotify_df.index.isin(training_rows)],
        metadata_df[metadata_df.index.isin(training_rows)]['genre'],
    ), (
        custom_df[~custom_df.index.isin(training_rows)],
        spotify_df[~spotify_df.index.isin(training_rows)],
        metadata_df[~metadata_df.index.isin(training_rows)]['genre'],
    )
```

```python
(train_custom, train_spotify, train_target), (test_custom, test_spotify, test_target) = split_data()
```

```python
display(train_custom, train_spotify, train_target, test_custom, test_spotify, test_target)
```

index	bpm	duration	chroma_cqt 0 0 min	chroma_cqt 0 0 max	chroma_cqt 0 0 median	chroma_cqt 0 0 mean	chroma_cqt 0 0 std	chroma_cqt 0 0 var	chroma_cqt 0 1 min	chroma_cqt 0 1 max	...	spectral_flatness 14 0 median
28375	0.609375	0.113878	0.000000	1.000000	0.104455	0.148400	0.143279	0.068509	0.000000	1.0	...	0.007854
29351	0.609375	0.008121	0.056375	0.080215	0.036457	0.052813	0.039562	0.016417	0.079502	1.0	...	0.000340
29550	0.922840	0.179235	0.000000	1.000000	0.537359	0.660093	0.476509	0.324563	0.000000	1.0	...	0.179992
29870	0.722222	0.088359	0.518517	1.000000	0.644458	0.764192	0.422127	0.273539	0.276575	1.0	...	0.271691
29919	0.509804	0.092874	0.055630	1.000000	0.257928	0.349367	0.415381	0.267460	0.077433	1.0	...	0.115422
...	...	...	...	...	...	...	...	...	...	...	...	...
209472	0.509804	0.187775	0.083717	1.000000	0.235423	0.306906	0.341179	0.204262	0.125626	1.0	...	0.004579
210784	0.342105	0.276565	0.294954	1.000000	0.398605	0.572311	0.650099	0.511541	0.089434	1.0	...	0.088479
219166	0.851190	0.139014	0.475242	1.000000	0.499780	0.637607	0.523679	0.371739	0.185524	1.0	...	0.198373

Рисунок 2.39 — Функція формування виборок

На цьому приготування даних завершені; можна приступати до побудови моделей. Далі моделі, сформовані на основі даних з датасету Spotify, будемо називати моделями Spotify, а моделі на основі власних метрик — власними.

РОЗДІЛ 3.
НАВЧАННЯ ТА ПОРІВНЯННЯ МОДЕЛЕЙ КЛАСИФІКАЦІЇ РІЗНИХ ВИДІВ

3.1. Метод найближчих сусідів

Клас sklearn.neighbors надає функціонал для методів навчання на основі сусідів з учителем та без учителя [29]. Метод найближчих сусідів в навчанні без учителя є основою багатьох інших методів, наприклад, багаторазового навчання та спектральної кластеризації. Навчання з учителем має два види: класифікація для даних із дискретними мітками та регресія для даних із безперервними мітками.

Принцип, що лежить в основі методу найближчого сусідів, полягає в тому, щоб знайти певну конкретну кількість навчальних записів, найближчих за відстанню до нової точки, і передбачити мітку з них. Кількість вибірок може бути визначеною розробником константою (метод k найближчих сусідів) або змінюватися на основі локальної щільності точок (метод найближчих сусідів на основі радіусу). Загалом, відстань може бути будь-якою метричною мірою; стандартна евклідова відстань є найпоширенішим вибором. Методи на основі сусідів відомі як неузагальнюючі методи машинного навчання, оскільки вони просто «запам’ятовують» всі свої навчальні дані (можливо, трансформовані в структуру швидкого індексування, таку як дерево куль або k-вимірне дерево).

Незважаючи на свою простоту, метод найближчих сусідів успішно використовується у великій кількості проблем класифікації та регресії, включаючи опрацювання рукописного тексту та супутникових зображень. Будучи непараметричним методом, він часто виявляється успішним у задачах класифікації, коли межа рішення дуже нерегулярна.

Класи в sklearn.neighbors можуть обробляти або масиви NumPy, або матриці scipy.sparse. Для щільних матриць підтримується велика кількість

можливих метрик відстані. Для розріджених матриць, для пошуку використовуються довільні метрики Мінковського.

Існують численні алгоритми навчання, які покладаються на найближчих сусідів у своїй основі. Одним із прикладів є алгоритм оцінки щільності ядра.

Класифікація на основі сусідів відноситься до типу навчання на основі екземплярів, або навчання без узагальнення: вона не намагається побудувати загальну внутрішню модель, а просто зберігає екземпляри навчальних даних. Класифікація обчислюється на основі простої більшості голосів найближчих сусідів кожної точки: точці запиту призначається клас даних, який має найбільше представників у найближчих сусідах точки.

scikit-learn реалізує два різних класифікатори найближчих сусідів: KNeighborsClassifier реалізує навчання на основі k найближчих сусідів кожної точки запиту, де k — ціле значення, задане користувачем. RadiusNeighborsClassifier реалізує навчання на основі кількості сусідів у межах фіксованого радіусу кожної навчальної точки, де r — це значення з плаваючою комою, яке вказує користувач. Перший метод, k найближчих сусідів, є найбільш використовуваною технікою.

Оптимальний вибір значення k сильно залежить від даних: загалом, більше значення пригнічує вплив шуму, але робить межі класифікації менш чіткими.

У випадках, коли вибірка даних нерівномірна, класифікація сусідів на основі радіусу в класі RadiusNeighborsClassifier може бути кращим вибором. Вказується фіксований радіус, і, таким чином, точки в більш рідкісних околицях використовують менше найближчих сусідів для класифікації. Для просторів параметрів високої розмірності цей метод стає менш ефективним через так званого «прокляття розмірності».

Базова класифікація найближчих сусідів використовує однакові ваги: значення, присвоєне точці запиту, обчислюється на основі простої більшості голосів найближчих сусідів. За деяких обставин краще зважувати сусідів таким чином, щоб ближчі сусіди сприяли на рішення більшою мірою. Це

можна зробити за допомогою параметру weights. Значення за замовчуванням, weights = 'uniform', призначає однакові ваги кожному сусіду. weights = 'distance' призначає ваги, пропорційні відстані, оберненій відстані від точки запиту. Крім того, для обчислення ваг можна надати визначену кастомно функцію відстані.

Побудуймо модель на основі найближчих сусідів. Передусім необхідно імпортувати відповідний клас у бібліотеці.

Далі визначемо 2 функції для двох різних датасетів — так звані гральні майданчики. Вони необхідні для оптимізації рутинної роботи при підборі параметрів моделей. Ці функції дозволяють швидко перезапускати модель на різних налаштуваннях та виводять результати навчання.

Після цього можна запустити модель на параметрах за замовченням та переглянути оцінку результату класифікації тестувальної вибірки (останній рядок на рисунку).

```
In [65]: from sklearn.neighbors import KNeighborsClassifier

In [66]: neigh_results = {
             'custom': [],
             'spotify': [],
         }

In [67]: def neigh_playground_custom(*args, **kwargs):
             neigh = KNeighborsClassifier(*args, **kwargs)
             neigh.fit(train_custom, train_target)
             score = neigh.score(test_custom, test_target)
             neigh_results['custom'].append([{'score': score, 'args': args, 'kwargs': kwargs}])
             print('Custom metric score: ', score)

         def neigh_playground_spotify(*args, **kwargs):
             neigh = KNeighborsClassifier(*args, **kwargs)
             neigh.fit(train_spotify, train_target)
             score = neigh.score(test_spotify, test_target)
             neigh_results['spotify'].append([{'score': score, 'args': args, 'kwargs': kwargs}])
             print('Spotify metric score: ', score)

         Спробуємо оптимізувати кастомну.

In [68]: neigh_playground = neigh_playground_custom

In [79]: neigh_playground()

         Custom metric score:  0.2766884531590414
```

Рисунок 3.1 — Підготовка до роботи над моделлю найближчих сусідів

Для методу найближчих сусідів основним параметром є кількість сусідів; спробуємо змінити її та переглянути результат.

```
neigh_playground(n_neighbors=1)
neigh_playground(n_neighbors=3)
neigh_playground(n_neighbors=10)
neigh_playground(n_neighbors=20)
neigh_playground(n_neighbors=50)
```

```
Custom metric score:  0.22657952069716775
Custom metric score:  0.2570806100217865
Custom metric score:  0.2766884531590414
Custom metric score:  0.2657952069716776
Custom metric score:  0.1655773420479303
```

Рисунок 3.2 — Перевірка впливу кількості сусідів

Оптимальне знайдене значення — 10. Далі спробуємо змінити функцію вагів окремих точок на пропорційну до відстані:

```
neigh_playground(n_neighbors=1, weights='distance')
neigh_playground(n_neighbors=3, weights='distance')
neigh_playground(n_neighbors=10, weights='distance')
neigh_playground(n_neighbors=20, weights='distance')
neigh_playground(n_neighbors=50, weights='distance')
```

```
Custom metric score:  0.22657952069716775
Custom metric score:  0.23529411764705882
Custom metric score:  0.2679738562091503
Custom metric score:  0.2549019607843137
Custom metric score:  0.16993464052287582
```

Рисунок 3.3 — Результати пропорційних вагів

Помітної різниці немає. Тепер порівняємо різні значення степеню для метрики Мінковського.

```
neigh_playground(n_neighbors=10, weights='distance', p=1)
neigh_playground(n_neighbors=10, weights='distance', p=2)
neigh_playground(n_neighbors=10, weights='distance', p=5)
neigh_playground(n_neighbors=10, weights='distance', p=10)
neigh_playground(n_neighbors=10, weights='distance', p=20)
```

```
Custom metric score:  0.27233115468409586
Custom metric score:  0.2679738562091503
Custom metric score:  0.2657952069716776
Custom metric score:  0.20915032679738563
Custom metric score:  0.19389978213507625
```

Рисунок 3.4 — Зміна степеню в метриці відстані

Бачимо найбільшу ефективність у першого степеню. Також доцільно провести перевірку за рівномірних вагів.

```
neigh_playground(n_neighbors=10, p=1)
neigh_playground(n_neighbors=10, p=2)
neigh_playground(n_neighbors=10, p=5)
neigh_playground(n_neighbors=10, p=8)
neigh_playground(n_neighbors=10, p=10)
neigh_playground(n_neighbors=10, p=20)
```

```
Custom metric score:  0.2701525054466231
Custom metric score:  0.2766884531590414
Custom metric score:  0.28540305010893247
Custom metric score:  0.24836601307189543
Custom metric score:  0.2178649237472767
Custom metric score:  0.19389978213507625
```

Рисунок 3.5 — Зміна степеню на рівномірних вагах точок

Бачимо невеликий приріст за 5-го степеню. Ще один параметр, визначений в документації — leaf_size. Перевіримо його вплив.

```
neigh_playground(n_neighbors=10, p=8, leaf_size=5)
neigh_playground(n_neighbors=10, p=8, leaf_size=10)
neigh_playground(n_neighbors=10, p=8, leaf_size=20)
neigh_playground(n_neighbors=10, p=8, leaf_size=50)
neigh_playground(n_neighbors=10, p=8, leaf_size=100)
```

```
Custom metric score:  0.24836601307189543
Custom metric score:  0.24836601307189543
Custom metric score:  0.24836601307189543
Custom metric score:  0.24836601307189543
Custom metric score:  0.24836601307189543
```

Рисунок 3.6 — Перевірка зміни розміру листа

Різниці немає. Перевіримо також на моделі за замовченням:

```
neigh_playground(leaf_size=5)
neigh_playground(leaf_size=10)
neigh_playground(leaf_size=20)
neigh_playground(leaf_size=50)
neigh_playground(leaf_size=100)
```

```
Custom metric score:  0.2549019607843137
Custom metric score:  0.2549019607843137
Custom metric score:  0.2549019607843137
Custom metric score:  0.2549019607843137
Custom metric score:  0.2549019607843137
```

Рисунок 3.7 — Розмір листа в стандартній моделі

Вплив знову відсутній. Згідно з документацією, класифікатор може використати один з декількох алгоритмів підбору найближчих сусідів; втім, для дуже розподілених даних, класифікатор завжди використовує алгоритм brute замість вибраного. leaf_size ж має вплив тільки на алгоритмах «ball_tree» та «kd_tree», тому і не чинить ефекту в даній задачі.

Підбір параметрів вручну являє довгу и рутинну роботу. Крім того, при ручному підборі можна пропустити оптимальне значення, оскільки воно може проявлятись на значеннях параметрів, які самі по окремості можуть становити меншу ефективність, тому будуть проігноровані.

В scikit-learn можна та рекомендується виконувати пошук в просторі гіперпараметрів за найкращою оцінкою крос-валідації. Будь-який параметр, наданий під час побудови оцінювача, може бути оптимізований таким чином.

У бібліотеці передбачено два загальних підходи до пошуку параметрів: GridSearchCV вичерпно розглядає всі комбінації параметрів, тоді як RandomizedSearchCV може вибирати задану кількість кандидатів із простору параметрів із заданим розподілом. Обидва ці інструменти мають аналоги HalvingGridSearchCV і HalvingRandomSearchCV, які виконують послідовний поділ та можуть виконувати пошук комбінації набагато швидше.

Розробимо новий «майданчик», з використанням пошуку гіперпараметрів. Перед імпортуванням класу пошуку з послідовним поділом, необхідно імпортувати модуль experimental.

Майданчик приймає назву датасету, словник значень параметрів та метод пошуку. За замовченням використовується метод випадкового пошуку з поділом.

```
from sklearn.model_selection import GridSearchCV
```

```
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingRandomSearchCV
```

```
def parameter_search(dataset_name, classifier, parameter_grid, method=None, method_params=None):
    if method is None:
        method = HalvingRandomSearchCV
        method_params = dict(
            aggressive_elimination=True,
        )
    search = method(classifier, parameter_grid, **method_params)
    if dataset_name == 'spotify':
        search.fit(train_spotify, train_target)
        display(search.score(test_spotify, test_target))
    elif dataset_name == 'custom':
        search.fit(train_custom, train_target)
        display(search.score(test_custom, test_target))
    else:
        raise ValueError(f'Unknown dataset {dataset_name}')
    display(search.best_params_)
    return search
```

Рисунок 3.8 — Функція пошуку оптимальних параметрів моделі

Функція виводить в консоль результат навчання на оптимальних параметрах, а також їх знайдені значення, та повертає об’єкт пошуку.

Спробуємо використати метод пошуку для виявлення оптимального числа сусідів та степеню відстані:

```
In [43]: parameters = {
             'n_neighbors': [1, 2, 5, 10, 15, 20, 25],
             'p': [1, 2, 5, 10, 15, 20, 25],
         }
         parameter_search('custom', KNeighborsClassifier(), parameters)

         0.22004357298474944

         {'p': 25, 'n_neighbors': 2}
```

Рисунок 3.9 — Пошук числа сусідів та степеню

Оцінка навчання гірша за отриману при пошуку вручну, але такий підхід займає набагато менше часу. Зважаючи на велику кількість моделей, які

передбачається навчати, використовуватись буде саме цей підхід. Для об'єктивної оцінки, саму цей результат будемо вважати отриманим для власного датасету. Він складає **0.22004357298474944.**

Після цього перевіримо передбачення для датасету Spotify. Спробуємо знайти оптимальні значення основних метрик:

```
parameters = {
    'n_neighbors': [1, 2, 5, 10, 20],
    'p': [1, 2, 5, 10, 20],
    'leaf_size': [1, 2, 5, 10, 20],
}
parameter_search('spotify', KNeighborsClassifier(), parameters)

0.2962962962962963

{'p': 2, 'n_neighbors': 2, 'leaf_size': 2}
```

Рисунок 3.10 — Пошук оптимальних параметрів для датасету Spotify

Можемо спробувати використати генератор для пошуку на більшому полі значень.

```
parameters = {
    'n_neighbors': list(range(25)),
    'p': list(range(25)),
    'leaf_size': list(range(25)),
}
parameter_search('spotify', KNeighborsClassifier(), parameters)

0.25925925925925924

{'p': 5, 'n_neighbors': 14, 'leaf_size': 22}
```

Рисунок 3.11 — Пошук з використанням генератора

Бачимо зменшення точності. Воно викликано більшою кількістю можливих значень, серед яких складніше знайти результат: механізм пошуку з поділом діє тим точніше, чим менше поле значень, які потрібно перевірити, і чим більша навчальна вибірка.

Використаймо генератор для створення наборів значень, зосереджених довкола оптимальних, знайдених на першому кроці:

```
parameters = {
    'n_neighbors': list(range(5)),
    'p': list(range(5)),
    'leaf_size': list(range(15, 25)),
}
parameter_search('spotify', KNeighborsClassifier(), parameters)
```

```
0.35294117647058826

{'p': 2, 'n_neighbors': 1, 'leaf_size': 20}
```

Рисунок 3.12 — Результат пошуку серед значень, зосереджених довкола оптимальних

Значення степеню може бути дробним числом. Спробуємо застосувати масив дробних чисел з тими ж межами, що і на попередньому кроці:

```
parameters = {
    'n_neighbors': list(range(5)),
    'p': [i*0.1 for i in range(10, 30)],
    'leaf_size': list(range(15, 25)),
}
parameter_search('spotify', KNeighborsClassifier(), parameters)
```

```
0.3289760348583878

{'p': 1.1, 'n_neighbors': 3, 'leaf_size': 18}
```

Рисунок 3.13 — Пошук дробного значення степеню

Бачимо падіння ефективності. Інші параметри позитивного впливу не зчинять, тому можна вважати, що оптимальний результат Spotify складає **0.35294117647058826**.

3.2. Метод опорних векторів

Метод опорних векторів (Support vector machines) — це набір методів навчання з учителем, що використовуються для класифікації, регресії та виявлення викидів [30].

Перевагами методу опорних векторів є:

- Ефективний у просторах з високою кількістю розмірностей.

- Досить ефективний у випадках, коли кількість вимірів більше, ніж кількість зразків (релевантно для цієї задачі).
- Використовує підмножину навчальних точок у функції прийняття рішень (так звані опорні вектори), що є ефективним використанням пам’яті.
- Універсальний: для функції прийняття рішень можна задати різні функції ядра. Надаються ядра за замовченням, але також можна вказати власні.

До недоліків методу опорних векторів можна віднести:

- Якщо кількість властивостей набагато більше, ніж кількість зразків, уникайте перенавчання при підборі функцій ядра, і термін регуляризації має вирішальне значення.
- SVM не надають оцінки ймовірності безпосередньо, вони розраховуються за допомогою дорогої п’ятикратної перехресної перевірки.

Методи опорних векторів у scikit-learn підтримують як щільні (numpy.ndarray і конвертовані в numpy.asarray), так і розріджені (будь-який scipy.sparse) вибіркові вектори як вхідні дані. Однак, щоб використовувати SVM для прогнозування розріджених даних, він повинен бути правильним чином навчений. Для оптимальної продуктивності варто використовувати numpy.ndarray (для щільних даних) або scipy.sparse.csr_matrix (для розріджених) з dtype=float64.

Найпростіші параметри методу — кількість ітерацій та коефіцієнт регуляризації. Спробуємо знайти для них орієнтовні значення.

```
from sklearn.svm import LinearSVC
```

```
parameters = {
    'max_iter': list(range(5000, 15000, 1000)),
    'C': [0.01, 0.05, 0.1, 0.2, 0.5, 1],
}
parameter_search('custom', LinearSVC(), parameters)
```

```
0.3115468409586057

{'max_iter': 13000, 'C': 0.01}
```

Рисунок 3.14 — Пошук коефіцієнту регуляризації

Оптимальне обране значення коефіцієнту — мінімальне. Зсунемо шкалу на порядок менше:

```
parameters = {
    'max_iter': list(range(10000, 15000, 1000)),
    'C': [0.001, 0.002, 0.003, 0.005, 0.006, 0.007],
}
parameter_search('custom', LinearSVC(), parameters)
```

```
0.3093681917211329

{'max_iter': 13000, 'C': 0.007}
```

Рисунок 3.15 — Пошук регуляризації серед менших значень

Схоже, що вплив знайденого значення знаходиться в межах статистичної похибки.

Далі можна перевірити альтернативний метод пеналізації, тобто штрафу за перенавчання. За замовченням використовується метод «l2», для розріджених даних додатково пропонується метод «l1». При використанні цього методу необхідно також вказати обов’язковий параметр dual=False.

```
parameters = {
    'C': [0.001, 0.002, 0.003, 0.005, 0.006, 0.007],
}
parameter_search('custom', LinearSVC(penalty='l1', dual=False, **{'max_iter': 10000}), parameters)
```

```
0.1655773420479303

{'C': 0.002}
```

Рисунок 3.16 — Метод штрафування «l1»

Бачимо зниження точності. Доцільніше дотримуватись методу за замовченням.

Метод дозволяє змінювати функцію застосовану втрат. За замовченням використовується функція «squared_hinge», альтернатива їй — «hinge», тобто корінь з попередньої. Перевіримо її ефективність:

```
parameters = {
    'C': [0.001, 0.002, 0.003, 0.005, 0.006, 0.007],
}
parameter_search('custom', LinearSVC(loss='hinge', **{'max_iter': 10000}), parameters)
```

```
0.3115468409586057
{'C': 0.007}
```

Рисунок 3.17 — Результат з функцією втрат «hinge»

Різниця — на рівні статистичної похибки. Принципового впливу функція втрат не чинить.

Далі можна перевірити стратегію вирішення конфліктів. Для вирішення конфліктів (коли запис задовольняє декільком класам) використовується 2 підходи, «ovr» та «crammer_singer». Перший використовується за замовченням, перевіримо другий.

```
parameters = {
    'C': [0.001, 0.002, 0.003, 0.005, 0.006, 0.007],
}
parameter_search('custom', LinearSVC(loss='hinge', multi_class='crammer_singer', **{'max_iter': 10000}), parameters)
```

```
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Libli
near failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
```

```
0.2875816993464052
{'C': 0.005}
```

Рисунок 3.18 — Вплив алгоритму «crammer_singer»

Бачимо негативний результат, до того ж зростає вимога до числа ітерацій.

Наступний параметр — «intercept_scaling». Використовується для нормалізації даних в цьому методі. Знайдемо орієнтовне значення:

```
parameters = {
    'C': [0.001, 0.002, 0.003, 0.005, 0.006, 0.007],
    'intercept_scaling': list(range(20)),
}
parameter_search('custom', LinearSVC(loss='hinge', **{'max_iter': 10000}), parameters)
```

```
0.3159041394335512

{'intercept_scaling': 10, 'C': 0.002}
```

Рисунок 3.19 — Пошук оптимального intercept_scaling

Бачимо незначний приріст. Далі можна спробувати додати ваги класів. За замовченням класи мають однакову вагу, але можливо зробити її пропорційною їх представленні в навчальній вибірці.

```
parameters = {
    'C': [0.001, 0.002, 0.003, 0.005, 0.006, 0.007],
    'intercept_scaling': list(range(20)),
}
parameter_search('custom', LinearSVC(loss='hinge', class_weight='balanced', **{'max_iter': 10000}), parameters)
```

```
0.32679738562091504

{'intercept_scaling': 7, 'C': 0.005}
```

Рисунок 3.20 — Збалансовані ваги класів

Це додає один відсоток точності. Залишається перевірити останній параметр: «fit_intercept». Це логічний перемикач, що позначає використання intercept_scaling для централізації даних. За замовченням він включений, спробуємо встановити на False.

```
parameters = {
    'C': [0.001, 0.002, 0.003, 0.005, 0.006, 0.007],
    'intercept_scaling': list(range(20)),
}
parameter_search('custom', LinearSVC(loss='hinge', class_weight='balanced', fit_intercept=False, **{'max_iter': 10000}), par
```

```
0.3289760348583878

{'intercept_scaling': 14, 'C': 0.001}
```

Рисунок 3.21 — Вимкнений fit_intercept

Бачимо приріст оцінки, отже, зміна є доцільною.

Оптимальний результат власної моделі — **0.3289760348583878**. Далі знайдемо класифікацію за датасетом Spotify.

```
parameters = {
    'max_iter': list(range(5000, 15000, 1000)),
    'C': [0.01, 0.05, 0.1, 0.2, 0.5, 1],
}
parameter_search('spotify', LinearSVC(), parameters)
```

```
0.3572984749455338

{'max_iter': 13000, 'C': 0.1}
```

Рисунок 3.22 — Підбір початкових параметрів

Спробуємо наблизити значення регуляризації.

```
parameters = {
    'max_iter': list(range(5000, 15000, 1000)),
    'C': [0.01*i for i in range(7, 13)],
}
parameter_search('spotify', LinearSVC(), parameters)
```

```
0.3333333333333333

{'max_iter': 9000, 'C': 0.08}
```

Рисунок 3.23 — Більш точне значення регуляризації

Бачимо зменшення результатів. Варто перевірити інші параметри. Зокрема, перевіримо альтернативний метод штрафування.

```
parameters = {
    'C': [0.01*i for i in range(7, 13)],
}
parameter_search('spotify', LinearSVC(penalty='l1', dual=False, max_iter=13000), parameters)
```

```
0.2113289760348584

{'C': 0.08}
```

Рисунок 3.24 — Метод штрафування «l1» для Spotify

Можливо, варто скорегувати значення попередньо знайдених параметрів.

```
parameters = {
    'C': [0.01, 0.05, 0.1, 0.2, 0.5, 1],
}
parameter_search('spotify', LinearSVC(penalty='l1', dual=False, max_iter=13000), parameters)
```

```
0.48148148148148145

{'C': 1}
```

Рисунок 3.25 — Зміна регуляризації при штрафуванні «l1»

Бачимо ризький ріст ефективності. Спробуймо знайти точніше значення коефіцієнту.

```
parameters = {
    'C': [0.1*i for i in range(8, 13)],
}
parameter_search('spotify', LinearSVC(penalty='l1', dual=False, max_iter=13000), parameters)
```

```
0.48148148148148145

{'C': 1.0}
```

Рисунок 3.26 — Пошук точного значення К регуляризації

Приріст відсутній. Продовжимо перебирати параметри.

```
parameters = {
    'C': [0.1*i for i in range(8, 13)],
}
parameter_search('spotify', LinearSVC(loss='hinge', max_iter=13000), parameters)
```

```
0.4400871459694989

{'C': 0.8}
```

Рисунок 3.27 — Зміна методу обробки втрат

```
parameters = {
    'C': [0.1*i for i in range(8, 13)],
}
parameter_search('spotify', LinearSVC(penalty='l1', dual=False, multi_class='crammer_singer', max_iter=13000), parameters)
```

```
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Libli
near failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Libli
near failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Libli
near failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Libli
near failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Libli
near failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarning: Libli
near failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "

0.4117647058823529

{'C': 0.9}
```

Рисунок 3.28 — Зміна стратегії вирішення конфліктів

Бачимо негативну динаміку, цими параметрами доцільно знехтувати. Знайдемо оптимальний intercept_scaling:

```
parameters = {
    'C': [0.1*i for i in range(8, 13)],
    'intercept_scaling': list(range(20)),
}
parameter_search('spotify', LinearSVC(penalty='l1', dual=False, max_iter=13000), parameters)

0.4596949891067538

{'intercept_scaling': 18, 'C': 0.8}
```

Рисунок 3.29 — Пошук коефіцієнту intercept_scaling

Отриманий вплив — негативний. Спробуймо змінити стратегію зваження класів.

```
parameters = {
    'C': [0.1*i for i in range(8, 13)],
}
parameter_search('spotify', LinearSVC(penalty='l1', dual=False, max_iter=13000, class_weight='balanced'), parameters)

0.42919389978213507
```

Рисунок 3.30 — Збалансовані ваги класів

Залишається перевірити перемикач нормалізації:

```
parameters = {
    'C': [0.1*i for i in range(8, 13)],
}
parameter_search('spotify', LinearSVC(penalty='l1', dual=False, max_iter=13000, fit_intercept=False), parameters)
```

```
0.4662309368191721
{'C': 0.8}
```

Рисунок 3.31 — Результат вимкнутої внутрішньої нормалізації

Останні зміні виявились недоречними. Результат Spotify вважаємо **0.48148148148148145**.

3.3. Метод опорних векторів РБФ

Наступний метод класифікації — **метод опорних векторів РБФ**. Це метод опорних векторів, в якому ядром є радіальна базисна функція. РБФ — це дійсна функція, значення якої залежить від відстані між даною точкою та точкою початку координат, або будь-якою іншою сталою точкою, що називається центром. Функції РБФ можна використовувати як функції активації нейронних мереж та інших методів машинного навчання; крім того, лінійні комбінації радіальних базисних функцій можна використовувати для апроксимації інших функцій [31].

Під час навчання SVM з ядром Radial Basis Function необхідно враховувати два параметри: C, тобто значення регуляризації, і gamma. Параметр C, загальний для всіх ядер SVM, встановлює міру між правильністю класифікації навчальних прикладів із простотою поверхні прийняття рішень. Низький C робить поверхню рішення гладкою, а високий C спрямований на правильну класифікацію всіх навчальних прикладів. Параметр gamma визначає, який за величиною вплив має кожен навчальний приклад. Чим більша гама, тим ближче мають бути інші приклади, щоб заданий міг чинити вплив.

Традиційно, починаємо з моделі на власному датасеті. Спершу спробуємо підібрати коефіцієнт регуляризації.

```
from sklearn.svm import SVC
```

```
parameters = {
    'C': [0.01, 0.1, 0.5, 2, 5, 10, 25, 50, 100],
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.24183006535947713

{'C': 2}
```

Рисунок 3.32 — Орієнтовний пошук регуляризації

Бачимо, що знайдений коефіцієнт є натуральним числом; продовжуємо пошук в цьому напрямку.

```
parameters = {
    'C': [1, 2, 5, 10, 25, 50, 100],
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.3093681917211329

{'C': 50}

HalvingRandomSearchCV(estimator=SVC(),
                      param_distributions={'C': [1, 2, 5, 10, 25, 50, 100]},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'C': [20, 30, 40, 50, 60, 70, 80],
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.3093681917211329

{'C': 30}
```

Рисунок 3.33 — Пошук проміжку з оптимальним значенням регуляризації

Бажаний коефіцієнт знаходиться в проміжку 30; 50. Далі знайдемо значення гамми.

```
parameters = {
    'C': [30, 40, 50, 60, 70, 80],
    'gamma': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 'auto', 'scale']
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.15468409586056645

{'gamma': 0.5, 'C': 40}
```

Рисунок 3.34 — Пошук значення гамми за широкою шкалою

Перевіримо пошуком на вужчій шкалі.

```
parameters = {
    'C': [30, 40, 50, 60, 70, 80],
    'gamma': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 'auto', 'scale']
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.3093681917211329

{'gamma': 'scale', 'C': 40}

HalvingRandomSearchCV(estimator=SVC(),
                      param_distributions={'C': [30, 40, 50, 60, 70, 80],
                                           'gamma': [0.001, 0.005, 0.01, 0.05,
                                                     0.1, 0.5, 'auto',
                                                     'scale']},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'C': [30, 40, 50, 60, 70, 80],
    'gamma': [1, 5, 10, 50, 100, 'auto', 'scale']
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.15904139433551198

{'gamma': 100, 'C': 80}
```

Рисунок 3.35 — Звуження пошуку гамми

Натуральні значення не дають високого результату; серед дробних ж значень було обрано стандартне значення автоматичного визначення гамми. Варто порівняти два значення автоматичного визначення між собою.

```
parameters = {
    'C': [30, 40, 50, 60, 70, 80],
    'gamma': ['auto', 'scale']
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.3115468409586057

{'gamma': 'auto', 'C': 50}
```

Рисунок 3.36 — Вибір між стандартними механізмами розрахунку гамми

Бачимо, що найкращий результат було отримано саме функцією, будемо використовувати саме їх. Далі варто перевірити параметри скорочення навчання (оптимізує час) та пошуку вірогідності належності до класу.

```
parameters = {
    'C': [30, 40, 50, 60, 70, 80],
    'gamma': ['auto', 'scale'],
    'shrinking': [True, False],
    'probability': [True, False],
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.3093681917211329

{'shrinking': False, 'probability': False, 'gamma': 'scale', 'C': 50}
```

Рисунок 3.37 — Перевірка параметрів скорочення та використання вірогідності

Бачимо, що shrinking суттєвого впливу не чинить, probability залишається за замовченням. Спробуємо змінити коефіцієнт толерантності для навчання.

```
parameters = {
    'C': [30, 40, 50, 60, 70, 80],
    'gamma': ['auto', 'scale'],
    'tol': [10**(-i) for i in range(1, 10, 2)]
}
parameter_search('custom', SVC(kernel='rbf'), parameters)
```

```
0.3202614379084967

{'tol': 1e-09, 'gamma': 'auto', 'C': 80}
```

Рисунок 3.38 — Перевірка коефіцієнтів толерантності

Зменшення толерантності допомогло незначно збільшити результат. Він і є найкращим знайденим для власного датасету, і складає **0.3202614379084967**.

Далі опрацьовуємо Spotify. Шукаємо коефіцієнт регуляризації:

```
parameters = {
    'C': [0.01, 0.1, 0.5, 2, 5, 10, 25, 50, 100],
}
parameter_search('spotify', SVC(kernel='rbf'), parameters)
```

```
0.4226579520697168

{'C': 50}

HalvingRandomSearchCV(estimator=SVC(),
                      param_distributions={'C': [0.01, 0.1, 0.
                                                  50, 100]},
                      refit=<function _refit_callable at 0x000
```

```
parameters = {
    'C': [20, 30, 40, 50, 60, 70, 80],
}
parameter_search('spotify', SVC(kernel='rbf'), parameters)
```

```
0.4226579520697168

{'C': 40}
```

Рисунок 3.39 — Пошук регуляризації для Spotify

Оптимальне значення — в межах 40; 50. Далі знаходимо значення гамми, на загальній шкалі, потім — наближуємо.

```
parameters = {
    'C': [20, 30, 40, 50, 60, 70, 80],
    'gamma': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 'auto', 'scale']
}
parameter_search('spotify', SVC(kernel='rbf'), parameters)
```

```
0.4117647058823529

{'gamma': 1, 'C': 60}

HalvingRandomSearchCV(estimator=SVC(),
                      param_distributions={'C': [20, 30, 40, 50, 60, 70, 80],
                                           'gamma': [0.001, 0.005, 0.01, 0.05,
                                                     0.1, 0.5, 1, 5, 10, 50,
                                                     100, 'auto', 'scale']},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'C': [20, 30, 40, 50, 60, 70, 80],
    'gamma': [0.01, 0.05, 0.1]
}
parameter_search('spotify', SVC(kernel='rbf'), parameters)
```

```
0.4400871459694989

{'gamma': 0.1, 'C': 30}
```

Рисунок 3.40 — Пошук значення гамми з наближенням

Далі перевіряємо зменшення та пошук вірогідності.

```
parameters = {
    'C': [20, 30, 40, 50, 60, 70, 80],
    'gamma': [0.01, 0.05, 0.1],
    'shrinking': [True, False],
    'probability': [True, False],
}
parameter_search('spotify', SVC(kernel='rbf'), parameters)
```

```
0.4357298474945534

{'shrinking': True, 'probability': False, 'gamma': 0.05, 'C': 60}
```

Рисунок 3.41 — Пошук та вірогідність у Spotify

Впливу немає. Залишається перевірити коефіцієнт толерантності.

```
parameters = {
    'C': [20, 30, 40, 50, 60, 70, 80],
    'gamma': [0.01, 0.05, 0.1],
    'tol': [10**(-i) for i in range(1, 10, 2)]
}
parameter_search('spotify', SVC(kernel='rbf'), parameters)
```

```
0.46187363834422657

{'tol': 0.1, 'gamma': 0.1, 'C': 70}
```

Рисунок 3.42 — Пошук коефіцієнту толерантності

Як бачимо, високе значення коефіцієнту тут надає кращого результату. **0.46187363834422657** — значення для Spotify.

3.4. Гаусові процеси

Гаусові процеси (Gaussian Processes) — це вид методів навчання з учителем, призначений для вирішення проблем регресії та імовірнісної класифікації [32]. Перевагами Гаусових процесів є:

- Прогноз інтерполює спостереження (принаймні для стандартних ядер).
- Прогноз є імовірнісним (Гаусовим), тому можна обчислити емпіричні довірчі інтервали і на основі них прийняти рішення, чи слід перенавчити модель на іншій вибірці.
- Універсальність: можна використовувати різні види ядер. Scikit-learn надає стандартні ядра, але також можна створити власні ядра.

До недоліків Гаусових процесів можна віднести:

- Вони не є розрідженими, тобто вони використовують всю наявну інформацію про вибірку/метрики для виконання передбачення.
- Вони втрачають ефективність у просторах великих розмірів, а саме, коли кількість метрик перевищує кілька десятків.

Клас GaussianProcessClassifier реалізує процеси Гауса для цілей класифікації, точніше для імовірнісної класифікації, де тестові передбачення мають форму ймовірностей класів. GaussianProcessClassifier виконує Гаусові процеси перед прихованою (латентною) функцією f, яка потім проводиться через функцію зв’язку, щоб отримати імовірнісну класифікацію. Прихована функція f — це так звана неприємна функція, тобто така функція, значення якої не спостерігаються і самі по собі не є релевантними. Її мета полягає в тому, щоб забезпечити зручне формулювання моделі, і вона видаляється під час передбачення. GaussianProcessClassifier реалізує функцію логістичного зв’язку, для якої інтеграл не може бути обчислений аналітично, але легко апроксимується у двійковому випадку.

Перший параметр, який потрібно вирахувати — n_restarts_optimizer. Він позначає кількість перезапусків оптимізатора для пошуку параметрів ядра, які максимізують логарифмічні граничні ймовірності.

```
parameters = {
    'n_restarts_optimizer': [0, 1, 2, 5, 10, 20, 50, 100],
}
parameter_search('custom', GaussianProcessClassifier(), parameters)
```

```
0.21350762527233116

{'n_restarts_optimizer': 20}

HalvingRandomSearchCV(estimator=GaussianProcessClassifier(),
                      param_distributions={'n_restarts_optimizer': [0, 1, 2, 5,
                                                                    10, 20, 50,
                                                                    100]},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'n_restarts_optimizer': [50, 60, 80, 100],
}
parameter_search('custom', GaussianProcessClassifier(), parameters)
```

```
0.21350762527233116

{'n_restarts_optimizer': 60}

HalvingRandomSearchCV(estimator=GaussianProcessClassifier(),
                      param_distributions={'n_restarts_optimizer': [50, 60, 80,
                                                                    100]},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'max_iter_predict': [150, 200, 250, 300, 500],  # default: 100
}
parameter_search('custom', GaussianProcessClassifier(), parameters)
```

```
0.21350762527233116

{'max_iter_predict': 500}
```

Рисунок 3.43 — Пошук оптимальної кількості перезапусків

Бачимо, що суттєвої різниці між перезапусками немає. Крім того, можна змінити максимальна кількість ітерацій у методі Ньютона для апроксимації апостеріора під час прогнозування. Параметр warm_start ж може забезпечити обчислювальну оптимізацію. Знайдемо їх комбінацію.

```
parameters = {
    'n_restarts_optimizer': [0, 1, 2, 5, 10, 20, 50, 100],
    'max_iter_predict': [100, 150, 200, 250, 300, 500],  # default: 100
    'warm_start': [True, False],
}
parameter_search('custom', GaussianProcessClassifier(), parameters)
```

```
0.21350762527233116

{'warm_start': False, 'n_restarts_optimizer': 5, 'max_iter_predict': 200}
```

Рисунок 3.44 — Пошук кількості ітерацій в прогнозуванні та перевірка використання warm_start

Бачимо відсутність видимого впливу на результат; вказані параметри використовувати немає сенсу.

Останній параметр, що можна перевірити — стратегія конфліктів. За замовченням використовується «one_vs_rest», можемо застосувати альтернативну «one_vs_one».

```
parameters = {
    'n_restarts_optimizer': [0, 1, 2, 5, 10, 20, 50, 100],
    'max_iter_predict': [100, 150, 200, 250, 300, 500],  # default: 100
    'warm_start': [True, False],
}
parameter_search('custom', GaussianProcessClassifier(multi_class='one_vs_one'), parameters)
```

```
0.2549019607843137

{'warm_start': False, 'n_restarts_optimizer': 20, 'max_iter_predict': 250}
```

Рисунок 3.45 — Зміна стратегії вирішення конфліктів

Бачимо приріст в ефективності. Таким чином, найкращий знайдений результат власного датасету складає **0.2549019607843137**.

Тепер створюємо модель для Spotify. Розпочинаємо з пошуку кількості перезапусків оптимізатора. Як завжди, наближуємо масив кандидатів до найкращого попереднього значення.

```
parameters = {
    'n_restarts_optimizer': [0, 1, 2, 5, 10, 20, 50, 100],
}
parameter_search('spotify', GaussianProcessClassifier(), parameters)
```

```
0.3355119825708061

{'n_restarts_optimizer': 1}

HalvingRandomSearchCV(estimator=GaussianProcessClassifier(),
                      param_distributions={'n_restarts_optimizer': [0, 1
                                                                     10,
                                                                     100]
                      refit=<function _refit_callable at 0x0000022989B08
```

```
parameters = {
    'n_restarts_optimizer': [50, 60, 80, 100],
}
parameter_search('spotify', GaussianProcessClassifier(), parameters)
```

```
0.3355119825708061

{'n_restarts_optimizer': 100}

HalvingRandomSearchCV(estimator=GaussianProcessClassifier(),
                      param_distributions={'n_restarts_optimizer': [50,
                                                                     100]
                      refit=<function _refit_callable at 0x0000022989B08
```

```
parameters = {
    'max_iter_predict': [150, 200, 250, 300, 500],  # default: 100
}
parameter_search('spotify', GaussianProcessClassifier(), parameters)
```

```
0.3355119825708061

{'max_iter_predict': 250}
```

Рисунок 3.46 — Перевірка кількості перезапусків для Spotify

Бачимо, що вплив кількості перезапусків на результат відсутній. Це відповідає результатам і на попередній моделі. Перевіряємо наступні три параметри:

```
parameters = {
    'n_restarts_optimizer': [0, 1, 2, 5, 10, 20, 50, 100],
    'max_iter_predict': [100, 150, 200, 250, 300, 500],  # default: 100
    'warm_start': [True, False],
}
parameter_search('spotify', GaussianProcessClassifier(), parameters)
```

```
0.3355119825708061

{'warm_start': False, 'n_restarts_optimizer': 20, 'max_iter_predict': 100}

HalvingRandomSearchCV(estimator=GaussianProcessClassifier(),
                      param_distributions={'max_iter_predict': [100, 150, 200,
                                                                250, 300, 500],
                                           'n_restarts_optimizer': [0, 1, 2, 5,
                                                                    10, 20, 50,
                                                                    100],
                                           'warm_start': [True, False]},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'n_restarts_optimizer': [0, 1, 2, 5, 10, 20, 50, 100],
    'max_iter_predict': [100, 150, 200, 250, 300, 500],  # default: 100
    'warm_start': [True, False],
}
parameter_search('spotify', GaussianProcessClassifier(multi_class='one_vs_one'), parameters)
```

```
0.27450980392156865

{'warm_start': False, 'n_restarts_optimizer': 50, 'max_iter_predict': 150}
```

Рисунок 3.47 — Перевірка кількості ітерацій передбачення, «теплого старту» та конфліктної стратегії для Spotify

На відміну від моделі на власному датасеті, в моделі Spotify зміна стратегії вирішення конфліктів чинить негативний вплив. Отже, результатом моделі Spotify вважаємо **0.3355119825708061**.

3.5. Дерево рішень

Дерево рішень (DT) — це непараметричний метод навчання з учителем, який використовується для класифікації та регресії. Мета його полягає в тому, щоб створити модель, вивчаючи прості правила прийняття рішень, виведені з ознак даних. Дерево можна розглядати як постійну покрокову апроксимацію. Чим глибше дерево, тим складніші правила прийняття рішень, і тим краще модель [33].

Деякі переваги дерев рішень:

- Простота для розуміння та інтерпретації. Дерева можна візуалізувати.
- Вимагають малої підготовки даних. Інші методи часто вимагають нормалізації даних, створення фіктивних змінних і видалення пустих значень.
- Вартість використання дерева (тобто прогнозування даних) є логарифмичною функцією від кількості точок навчальних даних.
- Здатність обробляти як числові, так і категоріальні дані. Втім, поточна реалізація scikit-learn не підтримує категоріальні змінні.
- Здатність вирішувати задачі, що вимагають декілька виходів.
- Використання моделі білої коробки. Якщо певна ситуація спостерігається в моделі, вона легко пояснюється булевою логікою. Навпаки, у моделі чорного ящика (наприклад, у штучній нейронній мережі) результати можуть бути складнішими для інтерпретації.
- Можливість перевірити модель за допомогою статистичних тестів. Це дозволяє перевірити надійність моделі.
- Працюють добре, навіть якщо припущення дещо порушуються справжньою моделлю, на основі якої були згенеровані дані.

До недоліків дерев рішень можна віднести:

- Моделі, навчені деревами рішень можуть створювати надскладні дерева, які погано узагальнюють дані. Це називається перенавчанням. Щоб уникнути цієї проблеми, необхідні такі механізми, як обрізка, встановлення мінімальної кількості зразків, необхідних для листа дерева, встановлення максимальної глибини.
- Дерева рішень можуть бути нестабільними, оскільки невеликі варіації в даних можуть призвести до створення зовсім іншого

дерева. Ця проблема нівелюється використанням дерев рішень в ансамблі.

- Прогнози дерев рішень не є ні гладкими, ні безперервними, а дискретними наближеннями. Тому вони погано вміють екстраполювати.

Відомо, що проблема навчання оптимального дерева рішень є NP-повною за кількох аспектів оптимальності навіть для простих концепцій. Отже, практичні реалізації алгоритму дерева рішень засновані на евристичних алгоритмах, таких як жадібний алгоритм, де локально оптимальні рішення приймаються на кожному вузлі. Такі алгоритми не можуть гарантувати повернення глобально оптимального дерева рішень. Це можна компенсувати шляхом навчання кількох дерев у складі ансамблю, де спостереження та ознаки відбираються випадково.

Учні дерева рішень створюють упереджені дерева, якщо деякі класи кількісно домінують у навчальній вибірці. Тому рекомендується збалансувати набір даних перед навчанням з деревом рішень.

Переходимо до моделі на власному датасеті. Передусім встановлюємо 2 параметри: criterion, функція, що визначає якість розгалуження, та splitter, стратегія вибору напряму при розгалуженні.

```
parameters = {
    'criterion': ['gini', 'entropy'],
    'splitter': ['best', 'random'],
}
parameter_search('custom', DecisionTreeClassifier(), parameters)
```

```
0.2875816993464052

{'splitter': 'random', 'criterion': 'entropy'}
```

Рисунок 3.48 — Вибір параметрів criterion та splitter

Результат не відрізняється від стандартного. Перевіримо граничні параметри, що встановлюють мінімальну кількість спостережень для розгалуження та мінімальний обсяг листа.

```
parameters = {
    'min_samples_split': list(range(1, 6)),
    'min_samples_leaf': list(range(1, 6)),
}
parameter_search('custom', DecisionTreeClassifier(), parameters)
```

```
0.28104575163398693

{'min_samples_split': 5, 'min_samples_leaf': 4}
```

Рисунок 3.49 — Перевірка впливу граничних параметрів

Різниця в результаті негативна, правки недоцільні. Наступний параметр — кількість ознак, що беруться до уваги при пошуку найкращого розгалуження. За замовченням оцінюються всі ознаки.

```
parameters = {
    'max_features': [1, 2, 5, 10, 50, 100, 200, 500, 1000, 2000, 4000],
}
parameter_search('custom', DecisionTreeClassifier(), parameters)
```

```
0.2374727668845316

{'max_features': 5}
```

Рисунок 3.50 — Зміна кількості ознак при пошуці шляху

Результат знизився. Додатково було перевірено функцію підбору кількості ознак, результат також виявився незадовільним.

Залишається перевірити 2 параметра: min_impurity_decrease, граничне значення якості, що має бути подолане для потенційного розгалуження, а також ccp_alpha, що використовується для оптимізації обчислювальних витрат, обираючи піддерево, яке задовольняє заданому значенню.

```
parameters = {
    'min_impurity_decrease': [i*0.1 for i in range(100)],
}
parameter_search('custom', DecisionTreeClassifier(), parameters)
```

```
0.1568627450980392

{'min_impurity_decrease': 4.9}
```

Рисунок 3.51 — Перевірка мінімальної оптимізації точності

```
parameters = {
    'ccp_alpha': [i*0.1 for i in range(100)],
}
parameter_search('custom', DecisionTreeClassifier(), parameters)
```

```
0.1568627450980392

{'ccp_alpha': 8.200000000000001}
```

Рисунок 3.52 — Перевірка границі обчислювальної складності

Бачимо, що в цілому зміна параметрів для цієї моделі не дозволяє досягти більшої точності прогнозу. Результатом вважаємо значення для моделі за замовченням, **0.2875816993464052**. Перейдемо до моделі Spotify.

```
parameters = {
    'criterion': ['gini', 'entropy'],
    'splitter': ['best', 'random'],
}
parameter_search('spotify', DecisionTreeClassifier(), parameters)
```

```
0.4596949891067538

{'splitter': 'best', 'criterion': 'entropy'}
```

Рисунок 3.53 — Пошук оптимальних стратегій розгалуження

Перевіримо граничні значення об’єму дерева:

```
parameters = {
    'min_samples_split': list(range(1, 6)),
    'min_samples_leaf': list(range(1, 6)),
}
parameter_search('spotify', DecisionTreeClassifier(), parameters)
```

```
0.4444444444444444

{'min_samples_split': 3, 'min_samples_leaf': 1}
```

Рисунок 3.54 — Вплив альтернативних значень розміру піддерева та листа

Результат негативний. Можемо перевірити комбінацію стратегії та обсягу об’єму:

```
parameters = {
    'criterion': ['gini', 'entropy'],
    'splitter': ['best', 'random'],
    'min_samples_split': list(range(1, 6)),
    'min_samples_leaf': list(range(1, 6)),
}
parameter_search('spotify', DecisionTreeClassifier(), parameters)
```

```
0.40522875816993464

{'splitter': 'random',
 'min_samples_split': 2,
 'min_samples_leaf': 4,
 'criterion': 'entropy'}
```

Рисунок 3.55 — Пошук комбінації параметрів

Бачимо суттєву деградацію оцінки. Тепер оцінимо вплив кількості порівнюваних ознак в розгалуженні:

```
parameters = {
    'max_features': [1, 2, 5],
}
parameter_search('spotify', DecisionTreeClassifier(), parameters)
```

```
0.3442265795206972

{'max_features': 1}

HalvingRandomSearchCV(estimator=DecisionTreeClassifier(),
                      param_distributions={'max_features': [1, 2, 5]},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'max_features': ['auto', 'sqrt', 'log2'],
}
parameter_search('spotify', DecisionTreeClassifier(), parameters)
```

```
0.37037037037037035

{'max_features': 'sqrt'}
```

Рисунок 3.56 — Стала кількість та функції кількості ознак

Позитивна динаміка відсутня. Перевіримо два останні параметри:

```
parameters = {
    'min_impurity_decrease': [i*0.1 for i in range(100)],
}
parameter_search('spotify', DecisionTreeClassifier(), parameters)
```

```
0.1568627450980392

{'min_impurity_decrease': 0.4}

HalvingRandomSearchCV(estimator=DecisionTreeClassifier(),
                      param_distributions={'min_impurity_decrease': [(
```

```
parameters = {
    'ccp_alpha': [i*0.1 for i in range(100)],
}
parameter_search('spotify', DecisionTreeClassifier(), parameters)
```

```
0.1568627450980392

{'ccp_alpha': 2.8000000000000003}
```

Рисунок 3.57 — Граничні значення оптимізації точності та складності для Spotify

Бачимо такий самий негативний результат, як і в попередній моделі. Можемо зробити висновок про результат моделі Spotify, що складає **0.4596949891067538**.

3.6. Випадковий ліс

Random forest є ансамблевим методом прогнозування. Метою ансамблевих методів є поєднання прогнозів кількох базових оцінок, створених із заданим алгоритмом навчання, щоб покращити узагальнюваність/надійність

у порівнянні з однією оцінкою. Зазвичай виділяють два сімейства ансамблевих методів: усереднюючі та посилюючі. Випадковий ліс відноситься до першого виду.

У методах усереднення основним принципом є виконання декількох оцінок незалежно, а потім — їх усереднення. Комбінована оцінка зазвичай краща, ніж будь-яка однобазова оцінка, оскільки її дисперсія таким чином зменшується.

В scikit-learn, у випадкових лісах кожне дерево будується із вибірки, вилученої із заміною з навчального набору. Крім того, під час розбиття кожного вузла найкраще розщеплення виявляється або з усіх вхідних об’єктів, або з випадкової підмножини розміру max_features. Метою цих двох джерел випадковості є зменшення дисперсії оцінки лісу.

Дійсно, індивідуальні дерева рішень зазвичай демонструють високу дисперсію і мають тенденцію переповнюватися. Введена випадковість у лісах дає дерева рішень з дещо виокремленими помилками передбачення. Беручи середнє значення цих прогнозів, деякі помилки можна нівелювати. Випадкові ліси досягають зменшеної дисперсії шляхом поєднання різноманітних дерев, іноді за ціною невеликого збільшення упередженості. На практиці скорочення дисперсії часто є значним, отже, дає загальну кращу модель [34].

Приступаємо до розробки моделі на власних даних. Передусім знайдемо оптимальну кількість дерев:

```
parameters = {
    'n_estimators': [20, 30, 40, 50, 60],
}
parameter_search('custom', RandomForestClassifier(), parameters)
```

```
0.3464052287581699

{'n_estimators': 30}
```

Рисунок 3.58 — Пошук оптимальної кількості дерев після кроку наближення

Значення перевищує оцінку за замовченням. Далі було обчислено параметри окремих дерев. Втім, правки параметрів розщеплення та глибини не надають переваги в точності. Лише корегування максимальної кількості ознак в оцінці розгалуження надало оптимізацію результату:

```
parameters = {
    'max_features': [1000, 2000, 4000],
}
parameter_search('custom', RandomForestClassifier(), parameters)
```

```
0.33769063180827885
{'max_features': 1000}
```

Рисунок 3.59 — Оптимальне значення кількості оцінюваних ознак

Після цього було перевірено параметри дерева max_leaf_nodes, min_impurity_decrease, warm_start та ccp_alpha. Жоден з них не дав прогресу в точності класифікації.

Тепер перевіримо параметр bootstrap. Він дозволяє навчати окремі дерева на фрагментах тренувальної вибірки замість повної.

```
parameters = {
    'bootstrap': [True, False],
}
parameter_search('custom', RandomForestClassifier(max_features=1000), parameters)
```

```
0.3289760348583878
{'bootstrap': True}
```

Рисунок 3.60 — Перевірка навчання дерев на сабсетах

Бачимо позитивний приріст. Можемо перевірити використання повної вибірки для оцінки генералізації:

```
: parameters = {
    'oob_score': [True, False],
}
parameter_search('custom', RandomForestClassifier(max_features=1000, bootstrap=True), parameters)
```

```
0.32461873638344224
{'oob_score': True}
```

Рисунок 3.61 — Результат впливу перемикача oob_score

Наприкінці спробуймо перевірити разом всі параметри, які окремо не дали позитивного впливу; можливо, вони дадуть ефективний набор.

```
parameters = {
    'n_estimators': [20, 30, 40, 50, 60],
    'criterion': ['gini', 'entropy'],
    'max_depth': [1, 10, 50, 100, 200],
    'min_samples_split': list(range(1, 6)),
    'min_samples_leaf': list(range(2, 6)),
    'max_leaf_nodes': [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, None],
    'oob_score': [True, False],
    'warm_start': [True, False],
    'ccp_alpha': [i*0.1 for i in range(100)],
}
parameter_search('custom', RandomForestClassifier(max_features=1000, bootstrap=True), parameters)
```

```
0.15904139433551198

{'warm_start': False,
 'oob_score': False,
 'n_estimators': 30,
 'min_samples_split': 2,
 'min_samples_leaf': 2,
 'max_leaf_nodes': 20,
 'max_depth': 100,
 'criterion': 'entropy',
 'ccp_alpha': 9.0}
```

Рисунок 3.62 — Пошук оптимального набору попередніх

Припущення не виправдалось. Отож вважаємо результатом власного датасету **0.3464052287581699**, значення, отримане на початку. Тепер на черзі датасет Spotify.

```
parameters = {
    'n_estimators': [5, 10, 20, 50, 100, 200, 500],
}
parameter_search('spotify', RandomForestClassifier(), parameters)
```

```
0.37254901960784315

{'n_estimators': 5}

HalvingRandomSearchCV(estimator=RandomForestClassifier(),
                      param_distributions={'n_estimators': [5, 10, 20, 50, 100,
                                                            200, 500]},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'n_estimators': [200, 300, 400, 500, 600],
}
parameter_search('spotify', RandomForestClassifier(), parameters)
```

```
0.5533769063180828

{'n_estimators': 400}
```

Рисунок 3.63 — Пошук кількості дерев для Spotify

В результаті наближення оптимальну кількість дерев було визначено як 400. Далі — критерій якості розщеплення.

```
parameters = {
    'criterion': ['gini', 'entropy'],
}
parameter_search('spotify', RandomForestClassifier(n_estimators=400), parameters)

0.5642701525054467

{'criterion': 'gini'}
```

Рисунок 3.64 — Оцінка критерію розщеплення

Декілька спроб показали, що знайдена різниця незначна, на рівні статистичної похибки. Перевіримо обмеження кількості ознак при оцінці розгалуження:

```
parameters = {
    'max_features': [1, 2, 5, 8, 10],
}
parameter_search('spotify', RandomForestClassifier(n_estimators=400), parameters)

0.5729847494553377

{'max_features': 10}
```

Рисунок 3.65 — Обмеження оцінюваної кількості ознак

Невелика позитивна динаміка. Інші параметри дерева прогресу в точності не дають. Розлянемо навчання на сабсетах та параметр оцінки генералізації:

```
parameters = {
    'bootstrap': [True, False],
}
parameter_search('spotify', RandomForestClassifier(n_estimators=400, max_features=10), parameters)
```

```
0.46187363834422657

{'bootstrap': False}

HalvingRandomSearchCV(estimator=RandomForestClassifier(max_features=10,
                                                       n_estimators=400),
                      param_distributions={'bootstrap': [True, False]},
                      refit=<function _refit_callable at 0x0000022989B084C0>)
```

```
parameters = {
    'oob_score': [True, False],
}
parameter_search('spotify', RandomForestClassifier(n_estimators=400, max_features=10), parameters)
```

```
0.5642701525054467

{'oob_score': False}
```

Рисунок 3.66 — Вплив параметрів bootstrap та oob_score

bootstrap краще залишити за замовченням, oob_score не чинить помітного впливу.

```
parameters = {
    'warm_start': [True, False],
}
parameter_search('spotify', RandomForestClassifier(n_estimators=400, max_features=10), parameters)
```

```
0.5664488017429193

{'warm_start': False}
```

Рисунок 3.67 — Перевірка донавчання на попередніх результатах

Залишається лише перевірити сукупній результат відкинутих параметрів.

```
parameters = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [1, 10, 50, 100, 200],
    'min_samples_split': list(range(1, 6)),
    'min_samples_leaf': list(range(2, 6)),
    'max_leaf_nodes': [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, None],
    'oob_score': [True, False],
    'warm_start': [True, False],
}
parameter_search('spotify', RandomForestClassifier(n_estimators=400, max_features=10), parameters)
```

```
0.4422657952069717

{'warm_start': True,
 'oob_score': False,
 'min_samples_split': 5,
 'min_samples_leaf': 3,
 'max_leaf_nodes': 1000,
 'max_depth': 1,
 'criterion': 'entropy'}
```

Рисунок 3.68 — Оптимальний набір попередніх параметрів

Бачимо негативну динаміку, отже, цими значеннями доцільно знехтувати. Найкращим результатом для Spotify в випадковому лісі є **0.5729847494553377**.

3.7. Нейронна мережа (багатошаровий перцептрон)

Багатошаровий перцептрон (MLP) — це керований алгоритм навчання, який вивчає функцію $f(.): R^m \rightarrow R^o$, тренуючись на наборі даних, де m — розмірність вводу, та o — розмірність результату. Враховуючи набір ознак $X = x_1, x_2, \dots, x_m$ і цільовий показник y, він може навчитися нелінійній апроксимації функції як для класифікації, так і для регресії. Він відрізняється від логістичної регресії тим, що між вхідним і вихідним шаром може бути один або кілька нелінійних шарів, які називаються прихованими шарами [27].

Крайній лівий шар, або вхідний шар, складається з набору нейронів $\{x_i | x_1, x_2, \dots x_m\}$, що представляють вхідні ознаки. Кожен нейрон у прихованому шарі перетворює значення з попереднього шару зваженим лінійним підсумовуванням:

$$w_1 x_1 + w_2 x_2 + \dots + w_m x_m \qquad (3.1)$$

а потім застосовує нелінійну функцію активації $g(.): R \rightarrow R$, наприклад, гіперболічний тангенс. Вихідний рівень приймає значення з останнього прихованого рівня і перетворює їх у вихідні значення.

Перевагами багатошарового перцептрону є:

- Можливість вивчення нелінійних моделей.
- Можливість часткового вивчення моделей у режимі реального часу.

До недоліків багатошарового перцептрону належать:

- MLP із прихованими шарами мають неопуклу функцію втрат, коли існує більше одного локального мінімуму. Тому різні ініціалізації випадкової ваги можуть призвести до різної точності перевірки.

- MLP вимагає налаштування ряду гіперпараметрів, таких як кількість прихованих нейронів, шарів та ітерацій.
- MLP чутливий до масштабування функцій.

Приступаємо до моделі власних даних. Спершу для перцептрону необхідно визначити структуру шарів. Переглянемо структуру за замовченням.

```
from sklearn.neural_network import MLPClassifier
```

```
start_time = time.time()
custom_nn = MLPClassifier(hidden_layer_sizes=(100, ))
custom_nn.fit(train_custom, train_target)
print(f'Training took {time.time() - start_time}')
custom_nn.score(test_custom, test_target)
```

```
Training took 3.6887001991271973
```

```
f:\education\university\master\year2\diplom\src\venv\lib\
4: ConvergenceWarning: Stochastic Optimizer: Maximum itera
  warnings.warn(
```

```
0.3224400871459695
```

Рисунок 3.69 — Стандартна структура шарів перцептрону

Далі були перевірені числені формати структури, включаючи нерівномірні, зі зменшеною або збільшеною кількістю проміжкових шарів; «зростаючі» та «звужуючися». Лише дві з них дали позитивний результат.

```
start_time = time.time()
custom_nn = MLPClassifier(hidden_layer_sizes=(100, 100, 100, 100, 100, 100))
custom_nn.fit(train_custom, train_target)
print(f'Training took {time.time() - start_time}')
custom_nn.score(test_custom, test_target)
```

```
Training took 2.9406299591064453
```

```
0.3289760348583878
```

```
start_time = time.time()
custom_nn = MLPClassifier(hidden_layer_sizes=(1000, 1000, 1000))
custom_nn.fit(train_custom, train_target)
print(f'Training took {time.time() - start_time}')
custom_nn.score(test_custom, test_target)
```

```
Training took 27.883750915527344
```

```
0.3289760348583878
```

Рисунок 3.70 — Результати експериментів над структурою

Втім, цей результат знаходиться в межах статистичної похибки. Їм можна знехтувати, і використати структуру за замовченням, тобто один шар зі 100 нейронів.

Можна спробувати знайти більш ефективну функцію активації.

```
parameters = {
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
}
parameter_search('custom', MLPClassifier(hidden_layer_sizes=(100), max_iter=5000), parameters, method=GridSearchCV)
0.3006535947712418
{'activation': 'logistic'}
GridSearchCV(estimator=MLPClassifier(hidden_layer_sizes=100, max_iter=5000),
             param_grid={'activation': ['identity', 'logistic', 'tanh',
                                        'relu']})
```

```
parameters = {
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
}
parameter_search('custom', MLPClassifier(hidden_layer_sizes=(100)), parameters)
0.28322440087145967
{'activation': 'relu'}
```

Рисунок 3.71 — Підбір функції активації

Підбір, виконаний різними методами, позитивного результату не дав. Варто перевірити перебір комбінацій функції активації та функції зважування зв’язків.

```
parameters = {
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['lbfgs', 'sgd', 'adam'],
}
parameter_search('custom', MLPClassifier(hidden_layer_sizes=(100), max_iter=1000), parameters, method=HalvingRandomSearchCV,
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:61
4: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (1000) reached and the optimization hasn't converged yet.
  warnings.warn(
0.32679738562091504
{'solver': 'sgd', 'activation': 'tanh'}
```

Рисунок 3.72 — Пошук набору двох функцій моделі

Результат — на рівні похибки. Далі перевіримо значення критерію регуляризації.

```
parameters = {
    'alpha': [10**i for i in range(-10, 0)],
}
parameter_search('custom', MLPClassifier(hidden_layer_sizes=(100), max_iter=1000), parameters, method=HalvingRandomSearchCV,
0.3616557734204793
{'alpha': 1e-09}
```

Рисунок 3.73 — Пошук параметру регуляризації похибки

Бачимо позитивну динаміку; зберігаємо знайдене значення регуляризації. Тепер перевіримо вплив обсягу пакетів даних для стохастичного оптимізатора.

```
parameters = {
    'batch_size': list(range(10, 100, 10)),
}
parameter_search('custom', MLPClassifier(hidden_layer_sizes=(100), max_iter=1000, alpha=1e-09), parameters, method=HalvingRa
```

```
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:54
2: UserWarning: Got `batch_size` less than 1 or larger than sample size. It is going to be clipped
  warnings.warn("Got `batch_size` less than 1 or larger than "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:54
2: UserWarning: Got `batch_size` less than 1 or larger than sample size. It is going to be clipped
  warnings.warn("Got `batch_size` less than 1 or larger than "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:61
9: UserWarning: Training interrupted by user.
  warnings.warn("Training interrupted by user.")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:54
2: UserWarning: Got `batch_size` less than 1 or larger than sample size. It is going to be clipped
  warnings.warn("Got `batch_size` less than 1 or larger than "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:61
9: UserWarning: Training interrupted by user.
  warnings.warn("Training interrupted by user.")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:54
2: UserWarning: Got `batch_size` less than 1 or larger than sample size. It is going to be clipped
  warnings.warn("Got `batch_size` less than 1 or larger than "
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:54
2: UserWarning: Got `batch_size` less than 1 or larger than sample size. It is going to be clipped
  warnings.warn("Got `batch_size` less than 1 or larger than "
```

```
0.3812636165577342

{'batch_size': 50}
```

Рисунок 3.74 — Пошук значення batch_size

Незважаючи на позитивний результат на рисунку, подальша перевірка показала, що це був випадковий викид. Зрештою, обсяг пакетів був проігнорований.

Наступний параметр — темп навчання. Перевіримо різні значення окремо.

```
start_time = time.time()
MLPClassifier(hidden_layer_sizes=(100), max_iter=1000, alpha=1e-09, learning_rate='constant')
custom_nn.fit(train_custom, train_target)
print(f'Training took {time.time() - start_time}')
custom_nn.score(test_custom, test_target)
```

```
Training took 0.23800063133239746

0.25272331154684097
```

```
start_time = time.time()
MLPClassifier(hidden_layer_sizes=(100), max_iter=1000, alpha=1e-09, learning_rate='invscaling')
custom_nn.fit(train_custom, train_target)
print(f'Training took {time.time() - start_time}')
custom_nn.score(test_custom, test_target)
```

```
Training took 10.848796367645264

0.3572984749455338
```

```
start_time = time.time()
MLPClassifier(hidden_layer_sizes=(100), max_iter=1000, alpha=1e-09, learning_rate='adaptive')
custom_nn.fit(train_custom, train_target)
print(f'Training took {time.time() - start_time}')
custom_nn.score(test_custom, test_target)
```

```
Training took 9.36202621459961

0.3333333333333333
```

Рисунок 3.75 — Порівняння різних значень темпу навчання

Як бачимо, темп «invscaling» дає найкращий результат. Залишається перевірити значення початкового темпу навчання.

```
parameters = {
    'learning_rate_init': [10**i for i in range(-10, 0)],
}
parameter_search('custom', MLPClassifier(hidden_layer_sizes=(100), max_iter=1000, alpha=1e-09, learning_rate='invscaling'),
```

```
0.3616557734204793

{'learning_rate_init': 0.001}
```

Рисунок 3.76 — Зміна попереднього темпу навчання

Бачимо несуттєвий приріст, що пояснюється випадковою різницею.

Після цього було проведено декілька спроб знайти нову комбінацію параметрів функції активації та зважування з урахуванням раніше знайдених параметрів, але результати не зчинили позитивного зсуву. Таким чином, найкращим результатом з отриманим залишається **0.3812636165577342**.

Далі оброблюється датасет Spotify. Переглянемо результат за замовчуванням.

```
start_time = time.time()
spotify_nn = MLPClassifier(hidden_layer_sizes=(100, ))
spotify_nn.fit(train_spotify, train_target)
print(f'Training took {time.time() - start_time}')
spotify_nn.score(test_spotify, test_target)
```

```
Training took 0.1099996566772461
```

```
f:\education\university\master\year2\diplom\src\venv\li
4: ConvergenceWarning: Stochastic Optimizer: Maximum it
  warnings.warn(
```

```
0.47058823529411764
```

Рисунок 3.77 — Результат Spotify на стандартних параметрах

Аналогічно моделі на своєму датасеті, для Spotify була переглянута велика кількість варіантів структури. Одна з них дала досить помітний приріст, але він виявся спричинений випадковим фактором.

```
start_time = time.time()
spotify_nn = MLPClassifier(hidden_layer_sizes=(1000, 1000, 1000))
spotify_nn.fit(train_spotify, train_target)
print(f'Training took {time.time() - start_time}')
spotify_nn.score(test_spotify, test_target)
```

```
Training took 10.19378399848938

0.49019607843137253
```

Рисунок 3.78 — «Найкраща» структура

Зрештою, структуру знову було залишено за замовченням. Пошукаймо функції активації та зважування:

```
parameters = {
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['lbfgs', 'sgd', 'adam'],
}
parameter_search('spotify', MLPClassifier(hidden_layer_sizes=(100), max_iter=1000), parameters, method=HalvingRandomSearchCV
```

```
0.4596949891067538

{'solver': 'lbfgs', 'activation': 'relu'}
```

Рисунок 3.79 — Знайдені функції activation та solver

Результат негативний, тому ці функції залишаємо за замовченням. Далі, аналогічно «власній» моделі, було перевірено параметри alpha, batch_size,

learning_rate та learning_rate_init. Нажаль, жоден з них не дав приросту в ефективності класифікації.

```
parameters = {
    'learning_rate_init': [10**i for i in range(-10, 0)],
}
parameter_search('spotify', MLPClassifier(hidden_layer_sizes=(100), max_iter=1000), parameters, method=HalvingRandomSearchC\
```

```
0.46405228758169936
{'learning_rate_init': 0.1}
```

Рисунок 3.80 — Пошук оптимального початкового темпу навчання

Наприкінці було проведено повторний пошук функцій активації та зважування; нажаль, він знову не дав результатів. Зрештою, результатом Spotify можемо вважати отримане значення **0.49019607843137253**.

3.8. AdaBoost

AdaBoost — це один із методів ансамблювання, представлений в sklearn. Це алгоритм підсилювання, який був представлений у 1995 році Фройндом і Шапіром.

Основний принцип AdaBoost полягає в тому, щоб навчити послідовність слабких учнів (тобто моделей, які лише трохи кращі за випадкові припущення, наприклад, невеликі дерева рішень) на багаторазово змінених версіях даних. Прогнози з усіх них потім об’єднуються за допомогою зваженої більшості голосів (або суми) для отримання остаточного прогнозу.

Модифікації даних на кожній так званій ітерації підвищення полягають у застосуванні ваг до кожного з навчальних зразків. Спочатку всі ці ваги встановлені рівними, так що перший крок просто тренує учня на вихідних даних. Для кожної наступної ітерації ваги вибірки змінюються окремо, а алгоритм навчання повторно застосовується до перезважених даних. На кожному кроці ті навчальні приклади, які були неправильно передбачені посиленою моделлю на попередньому кроці мають збільшені ваги, тоді для тих, які були передбачені правильно — ваги зменшуються. З кожною ітерацією приклади, які важко передбачити, отримують все більший вплив.

Таким чином, кожен наступний слабкий учень змушений зосередитися на прикладах, пропущених попередніми в послідовності.

Класифікатор AdaBoost — це метаоцінювач, який спершу встановлює класифікатор до вихідного набору даних, а потім створює додаткові копії класифікатора до зміненого набору даних, в якому ваги неправильно класифікованих екземплярів коригуються таким чином, що наступні класифікатори більше зосереджуються на складні випадки [35].

Переходимо до моделі, навченої за AdaBoost на власному датасеті.

Ключовий параметр методу — base_estimator: тип моделі, що використовується на ітераціях підкріплення. Клас дозволяє вказувати різні види оцінювачів, але за замовченням використовує дерево рішень. Цей ж класифікатор буде застосований в даній моделі.

Наступні три параметри — кількість оцінювачів, швидкість навчання та алгоритм підсилювання. Їх можна підібрати у сукупності.

Було виконано 5 спроб підібрати набір параметрів, серед них було обрано найефективніший.

```
parameters = {
    'n_estimators': list(range(1, 100)),
    'learning_rate': [0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50],
    'algorithm': ['SAMME', 'SAMME.R'],
}
parameter_search('custom', AdaBoostClassifier(), parameters, method=HalvingRandomSearchCV, method_params=dict(factor=1.5))
```

```
0.29193899782135074

{'n_estimators': 81, 'learning_rate': 0.1, 'algorithm': 'SAMME.R'}
```

Рисунок 3.81 — Найкращий знайдений набір параметрів

Інші параметри безпосередньо в AdaBoost не редагуються; таким чином, найкращий результат власної вибірки — **0.29193899782135074**.

Для Spotify було проведено аналогічну роботу, з вибором найкращого результату, який склав **0.3660130718954248**.

```
parameters = {
    'n_estimators': list(range(1, 100)),
    'learning_rate': [0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20],
    'algorithm': ['SAMME', 'SAMME.R'],
}
parameter_search('spotify', AdaBoostClassifier(), parameters, method=HalvingRandomSearchCV, method_params=dict(factor=1.5))
```

```
0.4400871459694989

{'n_estimators': 58, 'learning_rate': 0.002, 'algorithm': 'SAMME.R'}
```

Рисунок 3.82 — Параметри, знайдені для Spotify

3.9. Наївний баєсів класифікатор

Наївні методи Баєса — це набір алгоритмів навчання з учителем, заснованих на застосуванні теореми Баєса з «наївним» припущенням про умовну незалежність між кожною парою ознак за значенням змінної класу.

Різні наївні баєсівські класифікатори відрізняються головним чином за припущеннями, які вони роблять щодо розподілу ознаки над цільовою змінною.

Незважаючи на їх, на перший погляд, спрощені припущення, наївні баєсівські класифікатори ефективно працюють в багатьох реальних ситуаціях, таких як класифікація документів і фільтрація спаму. Вони вимагають невеликої кількості навчальних даних для оцінки необхідних параметрів.

Наївні баєсівські учні та класифікатори можуть бути надзвичайно швидкими порівняно з більш складними методами. Відокремлення розподілів умовних ознак класу означає, що кожен розподіл може бути незалежно оцінений як одновимірний розподіл. Це, у свою чергу, допомагає полегшити проблеми, пов’язані з прокляттям розмірності.

З іншого боку, хоча наївний Баєс відомий як гідний класифікатор, він, як відомо, є поганим оцінювачем, тому результати ймовірності predict_proba в scikit-learn не слід сприймати надто серйозно [52].

В scikit-learn метод Баєса може бути використаний для класифікації. На вхід приймається лише два параметри: попередні вірогідності класів та коефіцієнт пом’якшення площини класифікації. Оптимальне значення першого параметру об’єкт визначає автоматично, тому для підбору залишається друге.

Модель на власних даних:

```
from sklearn.naive_bayes import GaussianNB
```

```
parameters = {
    'var_smoothing': [10**i for i in range(-20, -5)],
}
parameter_search('custom', GaussianNB(), parameters, method=HalvingRandomSearchCV, method_params=dict(factor=1.5))
```

```
0.22004357298474944

{'var_smoothing': 1e-20}
```

Рисунок 3.83 — Результат згладжування 1e-20

```
parameters = {
    'var_smoothing': [10**i for i in range(-20, -5)],
}
parameter_search('custom', GaussianNB(), parameters, method=HalvingRandomSearchCV, method_params=dict(factor=1.5))
```

```
0.22004357298474944
{'var_smoothing': 1e-18}
```

Рисунок 3.84 — Результат згладжування 1e-18

```
parameters = {
    'var_smoothing': [10**i for i in range(-20, -5)],
}
parameter_search('custom', GaussianNB(), parameters, method=HalvingRandomSearchCV, method_params=dict(factor=1.5))
```

```
0.22004357298474944
{'var_smoothing': 1e-07}
```

Рисунок 3.85 — Результат згладжування 1e-7

Бачимо, що коефіцієнт згладжування на результат впливу не чинить. Вважаємо результат власного датасету рівним **0.22004357298474944**. Переходимо до моделі Spotify.

```
parameters = {
    'var_smoothing': [10**i for i in range(-20, -5)],
}
parameter_search('spotify', GaussianNB(), parameters, method=HalvingRandomSearchCV, method_params=dict(factor=1.5))
```

```
0.3572984749455338
{'var_smoothing': 1e-10}
```

Рисунок 3.86 — Результат згладжування 1e-10

```
parameters = {
    'var_smoothing': [10**i for i in range(-20, -5)],
}
parameter_search('spotify', GaussianNB(), parameters, method=HalvingRandomSearchCV, method_params=dict(factor=1.5))
```

```
0.3355119825708061
{'var_smoothing': 1e-14}
```

Рисунок 3.87 — Результат згладжування 1e-14

```
parameters = {
    'var_smoothing': [10**i for i in range(-20, -5)],
}
parameter_search('spotify', GaussianNB(), parameters, method=HalvingRandomSearchCV, method_params=dict(factor=1.5))
```

```
0.3572984749455338
{'var_smoothing': 1e-10}
```

Рисунок 3.88 — Результат згладжування 1e-10 (повторний)

Бачимо, що коефіцієнт 1e-10 надав найкращий результат, і, до того ж, трапився в пошуку двічі. Можемо назвати результат Spotify **0.3572984749455338**.

3.10. Квадратичний дискримінантний аналіз

Квадратичний дискримінантний аналіз (Quadratic Discriminant Analysis) є нелінійним узагальненням методу лінійного дискримінантного аналізу, що не використовує припущення про однорідність коваріаційної матриці. Як вирішальне правило, застосовується квадратична функція над коваріаційною матрицею класу та апріорною вірогідністю спостереження [53].

Квадратичний дискримінантний аналіз дуже ефективний, коли поверхня, що розділяє класи, має яскраво виражений нелінійний характер (наприклад, параболоїд, або еліпсоїд у 3D-випадку). Однак він зберігає більшість недоліків LDA: використовує припущення про нормальність розподілу і не працює, коли матриці коваріацій вироджені (наприклад, при великій кількості змінних). Іншим недоліком QDA є те, що рівняння роздільної гіперповерхні виражене в неявному вигляді і не може бути використане для інтерпритації результатів [54].

В scikit-learn для виконання квадратичного дискримінантного аналізу використовується класифікатор з квадратичною межею рішення, створений шляхом підгонки умовних щільностей класів до даних і використання правила Байєса. Модель відповідає гауссовій щільності для кожного класу.

Клас QuadraticDiscriminantAnalysis приймає параметр priors, що позначає попередні ваги класів, але вони можуть бути визначені автоматично більш точно. Наступний параметр — коефіцієнт регуляризації класової коваріації. Спробуємо підібрати його для «власної» моделі.

```
parameters = {
    'reg_param': [0.01*i for i in range(-100, 100)],
}
parameter_search('custom', QuadraticDiscriminantAnalysis(), parameters, method=HalvingRandomSearchCV, method_params=dict(fac
```

```
g: Variables are collinear
  warnings.warn("Variables are collinear")

0.20043572984749455

{'reg_param': 0.8}
```

Рисунок 3.89 — Оптимальне значення коефіцієнту регуляризації

Отримане значення було винайдено в результаті пошуку, повтореного тричі. Далі варто знайти порогове значення впливу єдиного спостереження.

```
parameters = {
    'tol': [10**i for i in range(-20, -1)],
}
parameter_search('custom', QuadraticDiscriminantAnalysis(), parameters, method=HalvingRandomSearchCV, method_params=dict(fac

g: Variables are collinear
  warnings.warn("Variables are collinear")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\discriminant_analysis.py:808: UserWarnin
g: Variables are collinear
  warnings.warn("Variables are collinear")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\discriminant_analysis.py:808: UserWarnin
g: Variables are collinear
  warnings.warn("Variables are collinear")

0.16339869281045752

{'tol': 0.001}
```

Рисунок 3.90 — Значення впливу одного спостереження

Знайдене значення не впливає на результат за замовченням, тому може бути проігнороване. Отже, результат моделі на власних даних — **0.20043572984749455**.

Для Spotify пошук значень метапараметрів відбувається аналогічно.

```
parameters = {
    'reg_param': [0.01*i for i in range(-100, 100)],
}
parameter_search('spotify', QuadraticDiscriminantAnalysis(), parameters, method=HalvingRandomSearchCV, method_params=dict(fa

ing: invalid value encountered in log
  u = np.asarray([np.sum(np.log(s)) for s in self.scalings_])

0.16339869281045752

{'reg_param': -0.96}

HalvingRandomSearchCV(estimator=QuadraticDiscriminantAnalysis(), factor=1.5,
                      param_distributions={'reg_param': [-1.0, -0.99, -0.98,
                                                         -0.97, -0.96,
                                                         -0.9500000000000001,
                                                         -0.9400000000000001,
                                                         -0.93, -0.92, -0.91,
                                                         -0.9, -0.89, -0.88,
                                                         -0.87, -0.86, -0.85,
                                                         -0.84,
                                                         -0.8300000000000001,
                                                         -0.8200000000000001,
                                                         -0.81, -0.8, -0.79,
                                                         -0.78, -0.77, -0.76,
```

```
parameters = {
    'reg_param': [0.01*i for i in range(-100, 100)],
}
parameter_search('spotify', QuadraticDiscriminantAnalysis(), parameters, method=HalvingRandomSearchCV, method_params=dict(fa

0.16339869281045752

{'reg_param': -0.5}
```

Рисунок 3.91 — Пошук коефіцієнту регуляризації для Spotify

```
parameters = {
    'tol': [10**i for i in range(-20, -1)],
}
parameter_search('spotify', QuadraticDiscriminantAnalysis(), parameters, method=HalvingRandomSearchCV, method_params=dict(fa
```

```
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\discriminant_analysis.py:808: UserWarning:
Variables are collinear
  warnings.warn("Variables are collinear")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\discriminant_analysis.py:808: UserWarning:
Variables are collinear
  warnings.warn("Variables are collinear")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\discriminant_analysis.py:808: UserWarning:
Variables are collinear
  warnings.warn("Variables are collinear")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\discriminant_analysis.py:808: UserWarning:
Variables are collinear
  warnings.warn("Variables are collinear")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\discriminant_analysis.py:808: UserWarning:
Variables are collinear
  warnings.warn("Variables are collinear")
f:\education\university\master\year2\diplom\src\venv\lib\site-packages\sklearn\discriminant_analysis.py:808: UserWarning:
Variables are collinear
  warnings.warn("Variables are collinear")
```

```
0.16775599128540306
```

```
{'tol': 1e-11}
```

Рисунок 3.92 — Пошук оптимального значення впливу

Нажаль, для моделі на основі даних Spotify оптимальних параметрів знайдено не було, і результат залишається на рівні випадкового припущення, а саме — **0.16775599128540306**.

3.11. Аналіз результатів

Перед тим, як робити висновки щодо отриманих результатів, варто провести перевірку. В якості результатів нотувалося значення score моделі scikit; перевіримо, чи дійсно відповідає воно тестовій класифікації. Для цього навчимо одну модель та перевіримо, як саме вона класифікує об’єкти.

```
analysis = QuadraticDiscriminantAnalysis(**{'tol': 1e-11})
analysis.fit(train_spotify, train_target)
```

```
f:\education\university\master\year2\diplom\src\venv\lib\sit
Variables are collinear
  warnings.warn("Variables are collinear")
```

```
QuadraticDiscriminantAnalysis(tol=1e-11)
```

```
analysis.score(test_spotify, test_target)
```

```
0.1437908496732026
```

Рисунок 3.93 — Навчання певної моделі для перевірки

Отже, точність моделі оцінується як 0.1437908496732026. Перевіримо її: переглянемо результат передбачень та знайдемо частку правильних.

```
analysis.predict(test_spotify)
```

```
array(['Rap', 'Rap', 'Blues', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Anime', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'World', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Anime', 'Rap', 'Blues', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Anime', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Rap', 'Rap',
       'Pop', 'Blues', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Rap', 'Rap', 'Rap',
       'World', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Pop',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Pop', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Anime', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Pop', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Blues', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Pop', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Anime', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'World', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Blues', 'Rap', 'Rap', 'Rap',
       'Rap', 'Pop', 'Rap', 'Rap', 'Rap', 'World', 'Anime', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'World', 'Rap', 'World', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World', 'Anime',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'World',
       'World', 'Rap', 'Rap', 'Rap', 'Blues', 'Rap', 'Rap', 'Rap', 'Rap',
       'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap', 'Rap'],
      dtype=object)
```

Рисунок 3.94 — Передбачені лейбли в моделі для перевірки

Знайдемо кількість правильних передбачень.

```
test_target
```

```
27810     Anime
27878     Anime
28111     Anime
28118     Anime
28375     Anime
          ...
221292    World
221298    World
221315    World
221389    World
221630    World
Name: genre, Length: 459, dtype: object
```

```
prediction = analysis.predict(test_spotify)
print(sum(test_target == prediction), len(test_target), sum(test_target == prediction)/len(test_target))
```

```
66 459 0.1437908496732026
```

Рисунок 3.95 — Кількість та частка правильних передбачень

Як бачимо, правильно класифіковано 66 спостережень із 459; ця частка відповідає заявленій, отже, результати моделей вважаємо достовірними.

	custom	spotify
KNeighborsClassifier	0.220044	0.352941
LinearSVC	0.328976	0.481481
SVC_rbf	0.320261	0.461874
GaussianProcessClassifier	0.254902	0.335512
DecisionTreeClassifier	0.287582	0.459695
RandomForestClassifier	0.346405	0.572985
MLPClassifier	0.381264	0.490196
AdaBoostClassifier	0.291939	0.440087
GaussianNB	0.220044	0.357298
QuadraticDiscriminantAnalysis	0.200436	0.167756

Рисунок 3.96 — Результати навчання моделей в порядку їх створення

В цілому, результати навчання свідчать про високу складність задачі. Обрані класи відносяться до розглянутих пісень дещо умовно, оскільки не завжди корелюють зі звучанням, а саме на ньому заснований розроблений в роботі підхід.

Варто зазначити, що оскільки взагалі класів було 6, то рівень випадкового припущення складає 1/6, або 0.16(6). Можемо бачити, що хоча навчені моделі показують не надто високий результат, все ж переважна кількість показує ефективність більшу за випадкове припущення, тому говорити про безперспективність розглянутих підходів недоцільно.

Задля порівняння багатьох моделей якістю підбору метапараметрів було знехтувано задля кількості підходів; можливо, подальше дослідження методів найбільш вдалих моделей дасть приріст в результаті, особливо за застосування ансамблю.

	custom	spotify
MLPClassifier	0.381264	0.490196
RandomForestClassifier	0.346405	0.572985
LinearSVC	0.328976	0.481481
SVC_rbf	0.320261	0.461874
AdaBoostClassifier	0.291939	0.440087
DecisionTreeClassifier	0.287582	0.459695
GaussianProcessClassifier	0.254902	0.335512
KNeighborsClassifier	0.220044	0.352941
GaussianNB	0.220044	0.357298
QuadraticDiscriminantAnalysis	0.200436	0.167756

Рисунок 3.97 — Рейтинг моделей за результатом власного датасету

Бачимо, що власний підхід до цифрового представлення аудіофайлу показав найбільшу ефективність в комбінації з методами нейронної мережі, випадкового лісу та лінійного методу опорних векторів. Нейронна мережа показала найбільшу ефективність, оскільки найкраще здатна опрацьовувати дані з великим обсягом ознак.

	custom	spotify
RandomForestClassifier	0.346405	0.572985
MLPClassifier	0.381264	0.490196
LinearSVC	0.328976	0.481481
SVC_rbf	0.320261	0.461874
DecisionTreeClassifier	0.287582	0.459695
AdaBoostClassifier	0.291939	0.440087
GaussianNB	0.220044	0.357298
KNeighborsClassifier	0.220044	0.352941
GaussianProcessClassifier	0.254902	0.335512
QuadraticDiscriminantAnalysis	0.200436	0.167756

Рисунок 3.98 — Рейтинг за результатом моделі Spotify

Ці ж три методи показали найбільшу ефективність за використання метрик Spotify, щоправда, тут з помітним відривом найкращий результат дав метод випадкового лісу.

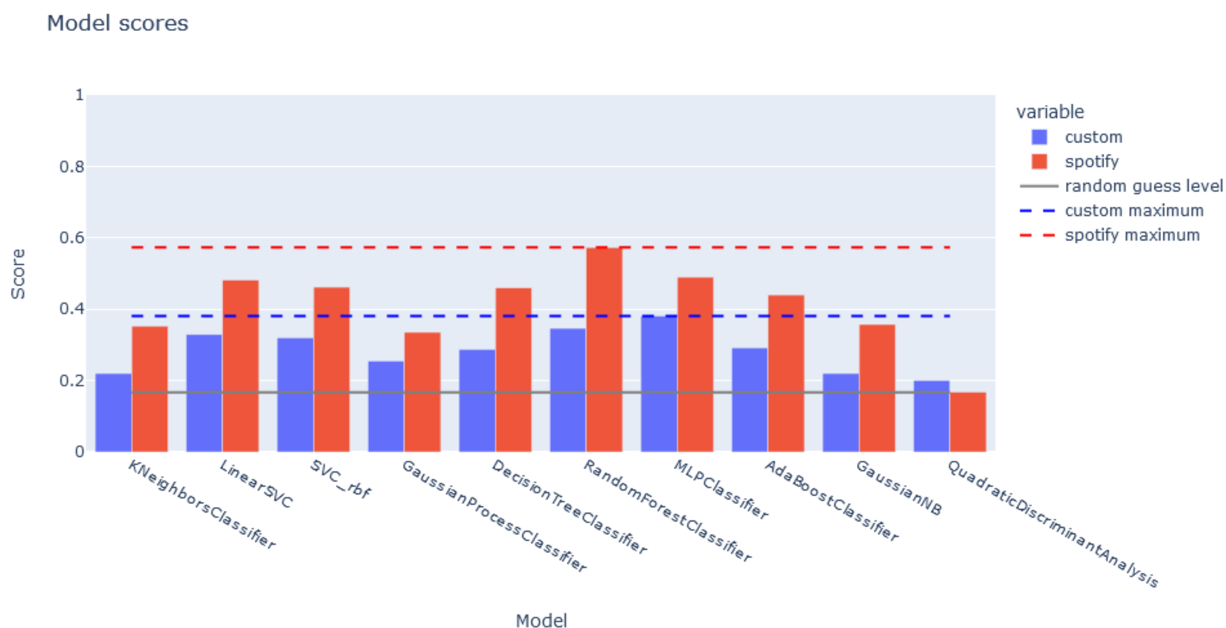


Рисунок 3.99 — Порівняльний графік всіх результатів, мінімального та максимального рівнів та рівню випадкового припущення

Власний підхід показав меншу ефективність класифікації, аніж метрики Spotify. Окрім можливих недоліків в виборі метрик, можна назвати ще декілька причин, які спричинили отриманий результат:

- Використаний набір даних містить надмірно високу кількість ознак (4514) при недостатній для такої розмірності кількості прикладів (523). При цьому тренувальна вибірка взагалі містить 64 приклади. Ефективність методів машинного навчання знаходиться в оберненій залежності від розмірності даних та в прямій — від кількості значень.
- Кількість класів, наявних в наборі даних, що дорівнювала 6, також була надмірною для даних такої розмірності та обсягу. Менша кількість класів за того ж обсягу вибірки спричинила б вищий результат. Крім того, класи заздалегідь не повністю корелювали з

метриками: наприклад, окремим класом був жанр музики з кінофільму, який за звучанням фактично може відноситись до будь-якого іншого класу.

- «Правильні» значення класів для спостережень були взяті нечесним чином, а саме — з датасету Spotify. Можливо, що цей жанр взагалі був визначений машинним чином з метрик, які вилучає цей сервіс, і насправді не відповідає дійсності. Для об'єктивного значення лейблу варто брати його з третього джерела.
- Скрапер, що використовується для завантаження файлів, може мати незадовільну точність пошуку через сторонній сервіс пошуку Google, і завантажувати невірний файл. Так, в скрипті було встановлено фільтр за тривалістю відео, що відсіяв декілька випадків (вірогідно, підбірка або альбом). Але залишаються численні інші можливі випадки, де, наприклад, інше відео виявилось більш популярним аніж цільова пісня, і тому було обране саме воно.

Повертаючись до вибору метрик: в рамках даної роботи був застосований підхід, подібний до розробленого попередньо, під час виконання курсової роботи. Відмінність полягала в тому, що замість поділу пісні на окремі фрейми та інтерпретації кожного фрейму як об'єкту спостережень, фіксована кількість фреймів та результати по кожному з них були розташовані як ознаки спостережень, а самими спостереженнями стали окремі пісні.

Задача, в рамках якої був розроблений минулий підхід, полягала у розробці нейронних мереж для класифікації пісень двох груп за виконавцем: тобто розмірність даних та класів була меншою, а об'єм спостережень залишився приблизно рівним. Нейронна мережа класифікувала кожен фрагмент пісні, а фінальний вердикт був модою передбачень за фрагментами.

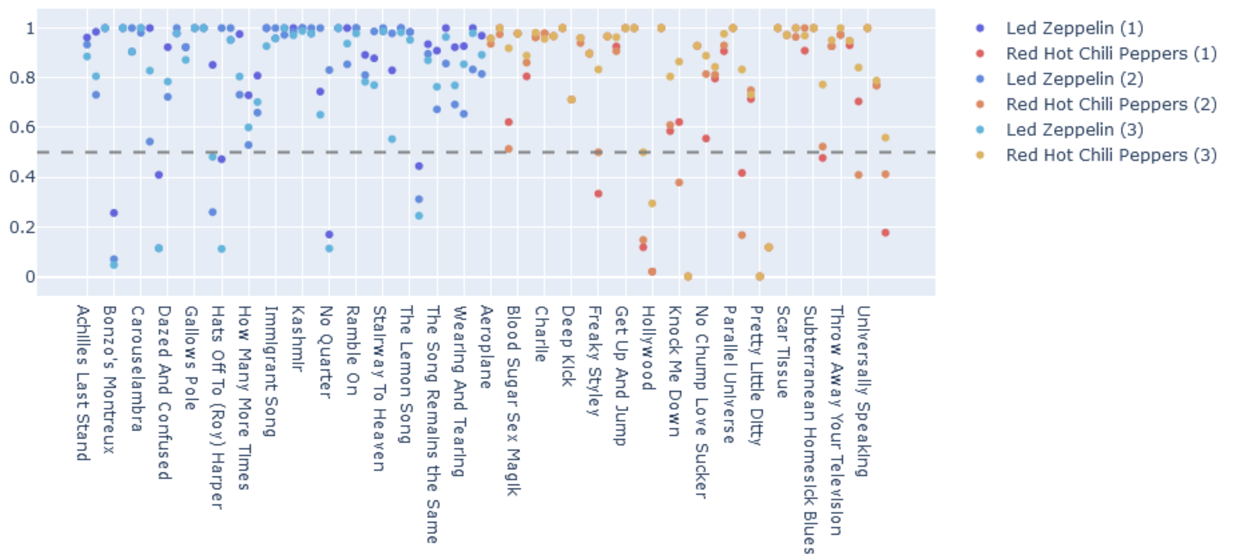


Рисунок 3.100 — Результат пошуку виконавця пісні за окремими фреймами

Як бачимо, точність такого підходу в тій задачі виявилась помітно вищою, аніж отримана в рамках дипломної роботи. Це може свідчити про недолік використаного підходу: вірогідно, більш ефективним способом визначення жанру буде поділ кожної пісні на фрагменти та класифікація пофреймово.

Складність тут полягає лише в інтерпретації результатів такої складеної моделі; в рамках роботи було навмисно розроблено саме такий формат даних, який дозволяє порівнювати результати з іншою двовимірною моделлю.

Результати щодо оптимальних алгоритмів, отримані в цій роботі, мають стати в нагоді при побудові більш точної складеної моделі класифікації аудіофайлів. Передбачається, що така модель буде поділяти кожну пісню на довільну кількість фреймів фіксованої довжини, використовувати ансамбль декількох методів машинного навчання для класифікації за кожним фреймом, та повертати вірогідність належності кожного запису до кожного класу (жанру або будь-якого іншого класифікатора).

ВИСНОВОК

Тема аналізу аудіосигналів надає широкий спектр задач для математики та акустики, і являє собою поле перетину цих дисциплін. Вирішення проблем цієї теми надає поштовх для розвитку теоретичної бази в цих двох науках.

Методи Machine Learning та Data Science сягають широкого застосування в аналізі аудіосигналів та музики, її властивостей — як сприйманих людиною, так и суто фізичних. Великі дані, що виникають під час цифрового представлення звуку та вилучення властивостей сигналу, часто вимагають використання спеціалізованих інструментів та підходів.

Для вирішення задач в рамках аналізу аудіосигналів розповсюджена практика вилучення певних властивостей таких сигналів. В виконаній роботі було проведено вилучення акустичних властивостей набору аудіосигналів, та побудовано набір даних з встановленим переліком їх характеристик.

Для адекватного представлення властивостей сигналу як входу моделі машинного навчання було розроблено ефективний алгоритм його перетворення. Алгоритм полягає у зменшенні розмірності шляхом розбиття на проміжки, та розрахунку статистик для кожної з акустичних властивостей. Для окремих властивостей, таких як BPM, застосовуються індивідуальні алгоритми нормалізації. З метою оцінювального порівняння, даний алгоритм було модифіковано для зменшення розмірності результату перетворення з трьовимірної до двовимірної матриці.

В рамках пошуку якнайточнішого алгоритму класифікації було розроблено та навчено велику кількість моделей, які загалом застосовують 10 різних алгоритмів. Шляхом підбору оптимальних метапараметрів, а також порівняння результатів для різних видів навчальних даних, було знайдено 3 оптимальні алгоритми для поставленої задачі. Ними виявились багатошаровий перцептрон, випадковий ліс та лінійний метод опорних векторів. Для моделей на розглянутих алгоритмах було підібрано оптимальні значення конфігурації, зокрема, структура мережі для перцептрону: одношарова зі 100 нейронами в

проміжковому шарі, що виявилась оптимальною як для зібраних даних, так і для контрольних.

В цілому, результат побудованих моделей в даній задачі виявився незадовільним. Такий результат було обґрунтовано впливом декількох факторів. Деякі з них зводяться до незадовільного формату навчальних даних: завеликої кількості ознак та класів при замалої кількості спостережень, недостатня відповідність тестувальних класів фактичним даним та їх заангажованість в бік контрольних моделей; недоліків в механізмі збору сирих даних перед їх цифровим представленням.

З метою пояснення результатів було проведене порівняння із задачею та результатами, отриманими при розробці попередньої версії алгоритму класифікації. Аналіз специфіки двох задач показав переваги і недоліки двох застосованих рішень; зважаючи на бажаний результат, було запропоновано альтернативний підхід для задачі класифікації аудіофайлів в загальному вигляді.

Такий підхід має зважати на сукупний результат, отриманий в рамках вирішення задач даної роботи та попередніх напрацювань. Передбачається, що він буде використовувати модель нового вигляду. Така модель буде поділяти кожну пісню на довільну кількість фреймів фіксованої довжини та вилучати показники, використовуючи попередню версію алгоритму. Далі необхідно використати ансамбль з декількох методів машинного навчання, які варто обрати серед найкращих знайдених в ході дипломної роботи, для класифікації за кожним фреймом. Результатом такої моделі буде набір вірогідностей належності кожного запису до кожного класу (жанру або будь-якого іншого класифікатора) за часткою належних фреймів.

Розробка описаного методу класифікації стане логічним продовженням та узагальненням результатів, отриманих в курсовій, дипломній та інших попередньо виконаних роботах на зміжну тему.

ПЕРЕЛІК ПОСИЛАНЬ

1. Machine Learning for Audio, Image and Video Analysis / F. Camastra, A. Vinciarelli. // Springer, 2015. 561 c.
2. Introduction to Audio Analysis A MATLAB Approach / T. Giannakopoulos, A. Pikrakis. // Academic Press, 2014. 288 c.
3. Data Mining: Concepts and Techniques / J. Han, M. Kamber, J. Pei // Morgan Kaufmann, 2012. 740 c.
4. MP3, AAC, WAV, FLAC: рассказываем обо всех форматах аудиофайлов [Електронний ресурс] https://www.audiomania.ru/ Режим доступу: https://www.audiomania.ru/content/art-7314.html (дата звернення 03.04.2021)
5. Quantitative Sound Analysis and the Visual Representations of Sound [Електронний ресурс] https://www.futurelearn.com/ Режим доступу: https://www.futurelearn.com/info/courses/music-moves/0/steps/12681 (дата звернення 03.04.2021)
6. Audio Data Analysis Using Deep Learning with Python (Part 1) [Електронний ресурс] https://www.kdnuggets.com/ Режим доступу: https://www.kdnuggets.com/2020/02/audio-data-analysis-deep-learning-python-part-1.html (дата звернення 18.04.2021)
7. PYO dedicated Python module for digital signal processing [Електронний ресурс] http://ajaxsoundstudio.com/ Режим доступу: http://ajaxsoundstudio.com/software/pyo/ (дата звернення 18.04.2021)
8. A Python library for audio feature extraction, classification, segmentation and applications [Електронний ресурс] https://github.com Режим доступу: https://github.com/tyiannak/pyAudioAnalysis/ (дата звернення 18.04.2021)
9. dejavu [Електронний ресурс] https://github.com/ Режим доступу: https://github.com/worldveil/dejavu (дата звернення 10.05.2021)
10. mingus [Електронний ресурс] https://github.com/ Режим доступу: https://github.com/chimezie/python3-mingus (дата звернення 10.05.2021)

11. hYPerSonic [Електронний ресурс] http://arrowtheory.com/ Режим доступу: http://arrowtheory.com/software/hypersonic/index.html (дата звернення 10.05.2021)
12. Pydub [Електронний ресурс] https://github.com/ Режим доступу: https://github.com/jiaaro/pydub (дата звернення 12.03.2021)
13. Loris [Електронний ресурс] https://github.com/ Режим доступу: https://github.com/tractal/loris (дата звернення 12.03.2021)
14. K. R. Fitz (1999). Reassigned Bandwidth-Enhanced Additive Sound Model. Завантажено з: https://www.researchgate.net/publication/243785607_The_reassigned_Bandwidth-Enhanced_Method_of_Additive_Synthesis
15. tinytag [Електронний ресурс] https://pypi.org/ Режим доступу: https://pypi.org/project/tinytag/ (дата звернення 12.03.2021)
16. Librosa [Електронний ресурс] https://librosa.org/ (дата звернення 12.03.2021)
17. M. Kattel, A. Nepal, A. K. Shah, D. Shrestha (2019). Chroma Feature Extraction. Завантажено з: https://www.researchgate.net/publication/330796993_Chroma_Feature_Extraction
18. Мел-кепстральные коэффициенты (MFCC) и распознавание речи [Електронний ресурс] https://habr.com/ Режим доступу: https://habr.com/ru/post/140828/ (дата звернення 16.03.2021)
19. D. Jang, M. Jin, C. D. Yoo (2008). MUSIC GENRE CLASSIFICATION USING NOVEL FEATURES AND A WEIGHTED VOTING METHOD. Завантажено з: https://slsp.kaist.ac.kr/paperdata/MUSIC_GENRE_CLASSIFICATION.pdf
20. D. N. Jiang, L. Lu, H. J. Zhang, J. H. Tao, L. H. Cai (2002). Music type classification by spectral contrast feature. Завантажено з: https://ieeexplore.ieee.org/document/1035731

21. C. Harte, M. Sandler, M. Gasser (2006). Detecting harmonic change in musical audio. Завантажено з: https://www.researchgate.net/publication/200806168_Detecting_harmonic_change_in_musical_audio
22. S. Dubnov (2004). Generalization of Spectral Flatness Measure for Non-Gaussian Linear Processes. Завантажено з: https://www.researchgate.net/publication/3343094_Generalization_of_Spectral_Flatness_Measure_for_Non-Gaussian_Linear_Processes
23. McFee, Brian, C. Raffel, D. Liang, D. PW Ellis, M. McVicar, E. Battenberg, and O. Nieto (2015). librosa: Audio and music signal analysis in python. Завантажено з: https://conference.scipy.org/proceedings/scipy2015/pdfs/brian_mcfee.pdf
24. librosa.util.frame [Електронний ресурс] https://librosa.org/ Режим доступу: https://librosa.org/doc/latest/generated/librosa.util.frame.html#librosa.util.frame (дата звернення 05.05.2021)
25. scikit-learn [Електронний ресурс] https://scikit-learn.org/ (дата звернення 12.05.2021)
26. 1.17. Neural network models (supervised) [Електронний ресурс] https://scikit-learn.org/ Режим доступу: https://scikit-learn.org/stable/modules/neural_networks_supervised.html (дата звернення 12.05.2021)
27. Multi-Layer Perceptron (MLP) [Електронний ресурс] https://xzz201920.medium.com/ Режим доступу: https://xzz201920.medium.com/multi-layer-perceptron-mlp-4e5c020fd28a (дата звернення 12.05.2021)
28. Intelligent Audio Analysis / B. W. Schuller // Springer, 2013. 345
29. Top Classification Algorithms Using Python [Електронний ресурс] https://www.analyticssteps.com Режим доступу: https://www.analyticssteps.com/blogs/top-classification-algorithms-using-python (дата звернення 20.05.2022)

30. How to Classify Data In Python using Scikit-learn [Електронний ресурс] https://www.activestate.com Режим доступу: https://www.activestate.com/resources/quick-reads/how-to-classify-data-in-python/ (дата звернення 20.05.2022)
31. Solving A Simple Classification Problem with Python — Fruits Lovers' Edition [Електронний ресурс] https://towardsdatascience.com Режим доступу: https://towardsdatascience.com/solving-a-simple-classification-problem-with-python-fruits-lovers-edition-d20ab6b071d2?gi=fbd282091f89 (дата звернення 20.05.2022)
32. Overview of Classification Methods in Python with Scikit-Learn [Електронний ресурс] https://stackabuse.com Режим доступу: https://stackabuse.com/overview-of-classification-methods-in-python-with-scikit-learn/ (дата звернення 20.05.2022)
33. Mastering Classification with Scikit-learn [Електронний ресурс] https://towardsdatascience.com Режим доступу: https://towardsdatascience.com/mastering-classification-with-scikit-learn-2cc0cffe33e3 (дата звернення 20.05.2022)
34. Что такое Scikit Learn - гайд по популярной библиотеке Python для начинающих [Електронний ресурс] https://datastart.ru Режим доступу: https://datastart.ru/blog/read/chto-takoe-scikit-learn-gayd-po-populyarnoy-biblioteke-python-dlya-nachinayuschih (дата звернення 20.05.2022)
35. Classifier comparison [Електронний ресурс] https://scikit-learn.org Режим доступу: https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html (дата звернення 20.05.2022)
36. How To Get a YouTube API Key (in 7 Simple Steps) [Електронний ресурс] https://rapidapi.com Режим доступу: https://rapidapi.com/blog/how-to-get-youtube-api-key/ (дата звернення 20.05.2022)

37. Youtube Search API [Електронний ресурс] https://www.npmjs.com Режим доступу: https://www.npmjs.com/package/youtube-search-api (дата звернення 20.05.2022)
38. Is there a Youtube API that gives only audio? [Електронний ресурс] https://stackoverflow.com Режим доступу: https://stackoverflow.com/questions/26548884/is-there-a-youtube-api-that-gives-only-audio (дата звернення 20.05.2022)
39. savefrom.net [Електронний ресурс] https://ru.savefrom.net Режим доступу: https://ru.savefrom.net/ (дата звернення 20.05.2022)
40. SaveDeo: Download videos online with ease! [Електронний ресурс] https://savedeo.site Режим доступу: https://savedeo.site/about (дата звернення 20.05.2022)
41. API Reference [Електронний ресурс] https://developers.google.com Режим доступу: https://developers.google.com/youtube/v3/docs (дата звернення 20.05.2022)
42. YouTube Data API (v3) - Quota Calculator [Електронний ресурс] https://developers.google.com Режим доступу: https://developers.google.com/youtube/v3/determine_quota_cost (дата звернення 20.05.2022)
43. How to Download Audio from YouTube: The Ultimate Guide [Електронний ресурс] https://www.parallels.com Режим доступу: https://www.parallels.com/tips/features/download-audio/ (дата звернення 20.05.2022)
44. youtube-dl [Електронний ресурс] https://github.com Режим доступу: https://github.com/ytdl-org/youtube-dl/tree/master/youtube_dl (дата звернення 20.05.2022)
45. Download the best quality audio file with youtube-dl [Електронний ресурс] https://stackoverflow.com Режим доступу: https://stackoverflow.com/questions/49804874/download-the-best-quality-audio-file-with-youtube-dl (дата звернення 20.05.2022)

46. NumPy Reference [Електронний ресурс] https://numpy.org Режим доступу: https://numpy.org/doc/stable/reference/ (дата звернення 20.05.2022)
47. User Guide [Електронний ресурс] https://pandas.pydata.org Режим доступу: https://pandas.pydata.org/docs/user_guide/ (дата звернення 20.05.2022)
48. Preprocessing data [Електронний ресурс] https://scikit-learn.org Режим доступу: https://scikit-learn.org/stable/modules/preprocessing.html (дата звернення 20.05.2022)
49. Как уменьшить количество измерений и извлечь из этого пользу [Електронний ресурс] https://habr.com Режим доступу: https://habr.com/ru/post/275273/ (дата звернення 20.05.2022)
50. Обзор нового алгоритма уменьшения размерности UMAP. Действительно ли он лучше и быстрее, чем t-SNE? [Електронний ресурс] https://habr.com Режим доступу: https://habr.com/ru/company/newprolab/blog/350584/ (дата звернення 20.05.2022)
51. Beautiful Soup Documentation [Електронний ресурс] https://www.crummy.com Режим доступу: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (дата звернення 20.05.2022)
52. Tony F. Chan, Gene H. Golub, Randall J. LeVeque (1979). Updating Formulae and a Pnirwise Algorithm for Computing Sample Variances. Завантажено з: http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf
53. Quadratic Discriminant Analysis [Електронний ресурс] https://towardsdatascience.com Режим доступу: https://towardsdatascience.com/quadratic-discriminant-analysis-ae55d8a8148a (дата звернення 20.05.2022)
54. Нелинейные классификаторы в R [Електронний ресурс] https://ranalytics.github.io Режим доступу: https://ranalytics.github.io/data-mining/074-Nonlinear-Classifiers.html (дата звернення 20.05.2022)