МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой № 43

| д-р техн. наук, профессор | | М.Ю. Охтилев |
|---|---|---|
| должность, уч. степень, звание | подпись, дата | инициалы, фамилия |

# БАКАЛАВРСКАЯ РАБОТА

на тему Определение эмоциональной окраски музыкальных произведений на основе анализа разнородных данных

выполнена Карповым Никитой Ивановичем
фамилия, имя, отчество студента в творительном падеже

по направлению подготовки 09.03.04 Программная инженерия
код наименование направления

наименование направления

направленности 02 Проектирование программных систем
код наименование направленности

наименование направленности

Студент группы № 4132 _____ Н.И. Карпов
подпись, дата инициалы, фамилия

Руководитель

| Доцент, к.т.н., доцент | | В.Ю. Скобцов |
|---|---|---|
| должность, уч. степень, звание | подпись, дата | инициалы, фамилия |

Санкт-Петербург 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

УТВЕРЖДАЮ
Заведующий кафедрой № 43

| д-р техн. наук, профессор | | М.Ю. Охтилев |
|---|---|---|
| должность, уч. степень, звание | подпись, дата | инициалы, фамилия |

# ЗАДАНИЕ НА ВЫПОЛНЕНИЕ БАКАЛАВРСКОЙ РАБОТЫ

студенту группы 4132 (номер) Карпову Никите Ивановичу (фамилия, имя, отчество)

на тему Определение эмоциональной окраски музыкальных произведений на основе анализа разнородных данных

утвержденную приказом ГУАП от 27.03.2025 № 11-387/25

Цель работы: разработка системы для автоматического определения эмоциональной окраски музыкальных произведений на основе анализа разнородных входных данных с применением методов машинного обучения

Задачи, подлежащие решению: анализ предметной области, анализ технологий, моделей, методов и инструментов, изучение и обработка набора данных, создание и тестирование модели машинного обучения, реализация веб-приложения

Содержание работы (основные разделы): введение, описание предметной области, анализ технологий для решения задачи, реализация программного продукта, заключение

Срок сдачи работы « 06 » июня 2025

Руководитель

| Доцент, к.т.н., доцент | 31.03.2025 | В.Ю. Скобцов |
|---|---|---|
| должность, уч. степень, звание | подпись, дата | инициалы, фамилия |

Задание принял(а) к исполнению
студент группы № 4132

| | 31.03.2025 | Н.И. Карпов |
|---|---|---|
| | подпись, дата | инициалы, фамилия |

## РЕФЕРАТ

Выпускная квалификационная работа, 147 страниц, 51 рисунок, 2 таблицы, 21 источник литературы, 7 приложений.

МАШИННОЕ ОБУЧЕНИЕ, ОБРАБОТКА СИГНАЛОВ, КЛАССИФИКАЦИЯ МУЗЫКИ, ГЛУБОКОЕ ОБУЧЕНИЕ, СЕТЕВЫЕ ТЕХНОЛОГИИ.

Цель работы – разработка системы для автоматического определения эмоциональной окраски музыкальных произведений на основе анализа разнородных входных данных с применением методов машинного обучения.

Объект исследования – процесс автоматического определения эмоциональной окраски музыкальных произведений.

Предмет исследования – методы и алгоритмы машинного обучения для классификации музыкальных произведений по эмоциональной окраске на основе анализа звукового сигнала, спектрограмм и текстовой составляющей.

В ходе работы было проведено исследование предметной области, технологических решений, исходных данных. Проведен сравнительный анализ разработанных методов и моделей, выбрана лучшая версия программного продукта.

В результате работы была создана модель глубокого обучения для классификации музыки. Модель была внедрена в клиент-серверное приложение.

# СОДЕРЖАНИЕ

## ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

API (Application Programming Interface) – интерфейс программирования приложений – набор правил для взаимодействия программ

CNN (Convolutional neural network) – сверточная нейронная сеть – архитектура глубоких искусственных нейронных сетей, нацеленная на распознавание образов

GPU (Graphic Processing Unit) – графический процессор – аппаратное устройство, оптимизированное для выполнения большого числа параллельных вычислений

GRU (Gated Recurrent Unit) – управляемый рекуррентный блок – архитектура глубоких искусственных нейронных сетей, предназначенная для обработки последовательностей

MLP (Multi-Layer Perceptron) – многослойный персептрон

Transformer – Трансформер – архитектура глубоких нейронных сетей, предназначенная для обработки длинных последовательностей на основе параллельных преобразований матриц и механизма внимания

Аугментация – искусственное расширение обучающей выборки путем применения преобразований к исходным данным

Алгоритм обратного распространения ошибки (Backpropagation) – метод оптимизации искусственных нейронных сетей, основанный на вычислении градиентов функции потерь по отношению к параметрам модели

Классификация – задача машинного обучения, в которой входной пример необходимо отнести к одному из заранее определенных классов

Функция ошибки (или функция потерь) – математическая функция, измеряющая степень расхождения между предсказаниями модели и ожидаемыми метками

# ВВЕДЕНИЕ

В последние годы наблюдается стремительный рост интереса к цифровой сфере развлечений, в частности, важным элементом досуга человека стало использование музыкальных платформ («Spotify», «Яндекс Музыка», «VK Музыка», «Звук» и других). Так, согласно данным исследования компании GfK от 2024 года, уже 47% россиян пользуется подпиской на музыкальные сервисы [1]. За первый квартал 2025 года база подписчиков «VK Музыки» возросла на 19%, и среднемесячная аудитория (не только подписчиков) составила 49 миллионов человек [2]. Среднемесячное число активных подписчиков «Яндекс Музыки» за первый квартал 2025 года составило 28 миллионов человек [3]. Одним из элементов успеха таких платформ является использование персонализированных рекомендательных систем, анализирующих жанровые предпочтения и поведение пользователя. Востребованной является возможность фильтрации музыки по ее эмоциональной окраске (настроению).

Для того чтобы рекомендательная система могла сопоставлять музыку с эмоциональным состоянием пользователя, каждая аудиозапись в каталоге должна быть снабжена набором меток, отражающих ее настроение. С учетом того, что объемы музыкальных баз превышают миллионы композиций, ручная разметка эмоциональных характеристик становится крайне трудоемкой и дорогой операцией, подверженной субъективным ошибкам. Это создает необходимость в разработке методов автоматического определения эмоциональной окраски музыкальных произведений.

Часть существующих методов опирается на характеристики музыки, которые невозможно получить только из аудиоданных. К ним относятся жанр, популярность, используемые инструменты, а также показатели, извлекаемые с помощью закрытых алгоритмов, такие как «valence» или «danceability» от Spotify.

Другие методы либо не предназначены для музыкальной аналитики, либо опираются на простые модели, чья точность оказывается недостаточной. В связи с этим актуально применение методов глубокого обучения, способных извлекать информативные признаки из сложных разнородных данных, например из звукового сигнала, спектрограмм или текста песен.

Целью работы является разработка системы для автоматического определения эмоциональной окраски музыкальных произведений на основе анализа разнородных входных данных с применением методов машинного обучения.

Для достижения цели необходимо решить следующие задачи:

1. Провести анализ предметной области и сформировать требования к системе;
2. Провести анализ современных технологий и выбрать необходимые для достижения цели методы, модели и инструменты;
3. Получить, изучить и провести предварительную обработку данных;
4. Создать и протестировать модели машинного обучения, сравнить результаты для разных архитектур и входных данных;
5. Разработать программное обеспечение для доступа к лучшей обученной модели.

Программный продукт ориентирован на открытость, точность автоматической разметки музыки, возможность использования без наличия любых данных, кроме аудиофайла.

# 1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1 Пользователи системы

Открытый интерфейс разрабатываемой системы предназначен для использования следующими сторонами:

1. Программное обеспечение музыкальных платформ – использует программный интерфейс для получения меток настроения и вероятностных распределений по эмоциональным классам;
2. Продуктовые менеджеры, аналитики и маркетологи – используют веб-приложение для выполнения предсказаний на одиночных примерах с визуализацией результата для составления музыкальных подборок.
3. Сторонние разработчики – имеют возможность загрузить обученную модель для ее внедрения в другие системы или для получения скрытых векторных представлений аудиоданных.

## 1.2 Характеристика решаемых задач и известные подходы

Задача определения эмоциональной окраски музыкальных произведений является задачей классификации: разрабатываемый алгоритм сопоставляет входящие данные с одной из заранее определенных меток (классов), например «грустный», «веселый», «энергичный», «спокойный». Расхождение между настоящим классом и полученной меткой называют ошибкой. Основной задачей работы является минимизация ошибки.

В качестве входных данных для алгоритма можно использовать признаки, которые нельзя извлечь из звукового сигнала, например жанр или используемые музыкальные инструменты. Но в таком случае применение программного продукта возможно только при наличии у пользователя всего набора заранее составленных признаков. Поэтому лучшим выбором входных данных для алгоритма является исходный звуковой сигнал, получаемый непосредственно из WAV или MP3 файлов.

Можно выделить два распространенных подхода к анализу аудиоданных:

1. Извлечение заранее определенных характеристик звукового сигнала. Чаще всего используются спектральные характеристики, например центр тяжести и ширина спектра, спектральные коэффициенты MFCC, или динамические признаки, например темп, квадрат амплитуды сигнала;

2. Анализ временных последовательностей: самого звукового сигнала, спектрограмм или текста композиции. В последнем случае также необходим алгоритм извлечения текста из аудиоданных.

Первый подход, позволяет решать задачу даже при помощи простых моделей. Однако строго ограниченный набор признаков упрощает представление исходного сигнала, что ведет к снижению точности алгоритма.

Второй подход накладывает ограничения на выбор моделей для решения задачи: анализ длинных многомерных последовательностей, таких как текст или спектрограммы, возможен только при использовании моделей глубокого обучения – многослойных моделей, обладающих возможностью строить сложные представления исходных данных [4, 5]. Конечно, подобные архитектуры повышают требования к аппаратным ресурсам и ведут к ухудшению объяснимости решений, принимаемых алгоритмом. Но с другой стороны, наличие «понимания» данных моделью с учетом локальных и глобальных шаблонов внутри них серьезно повышает качество предсказаний в условиях задачи обработки звукового сигнала, спектрограмм или текста.

Поскольку конечному пользователю в рамках данной задачи в большей степени важно качество классификации, а не объяснимость модели или ее сложность, предпочтение было отдано второму подходу с использованием моделей глубокого обучения.

## 1.3 Анализ требований

### 1.3.1 Требования к составу программного обеспечения

Для улучшения пользовательского опыта при использовании модели используется два приложения: клиентское и серверное. С помощью первого возможно использование модели для выполнения одиночных предсказаний с удобным пользовательским интерфейсом. Второе предназначено для программ и предусматривает возможность пакетной обработки аудиофайлов.

Таким образом, программное обеспечение состоит из нескольких модулей:

1. Модуль обучения моделей – предназначен для подготовки данных, построения, загрузки, обучения, оценки и сохранения моделей;
2. Модуль API модели – предоставляет интерфейс для доступа к обученной модели, построения предсказаний на одиночных примерах;
3. Серверный модуль – предоставляет интерфейс клиентскому приложению для доступа к API модели. Включает настройки политик доступа, перенаправления запросов и обработки входящих данных;
4. Клиентский модуль – предоставляет пользовательский интерфейс и удобный способ для взаимодействия с обученной моделью.

### 1.3.2 Функциональные требования

Программное обеспечение выполняет следующие функции:

1. Предварительная обработка данных – исходные данные обрабатываются с целью их очистки, нормализации и извлечения характеристик. В наборе данных очищаются целевые метки, происходит их слияние согласно алгоритму, описанному в разделе 3.4. Из звуковых файлов извлекаются необходимые характеристики. Спектрограммы и аудиоданные нормализуются, к ним применятся аугментация. Данные собираются в мини-пакеты и готовы к использованию при обучении моделей;

2. Обучение моделей – предварительно обработанные данные последовательно подаются на вход модели, и применяется оптимизатор. Модель оценивается на отложенной валидационной выборке и сохраняется в локальное хранилище;

3. Оценка моделей – обученные модели загружаются из локального хранилища и оцениваются на отложенной тестовой выборке. Данная функция необходима для критической оценки полученных моделей. Выполняется расчет метрик качества модели Precision, Recall, $F_1$-score, построение матрицы ошибок;

4. Запуск модели как сервиса и ее применение к новым пользовательским данным. Обученная модель размещается в локальной среде с предоставлением к ней API. Данный интерфейс позволяет запрашивать предсказания модели на одиночных примерах.

### 1.3.3 Требования к данным

Система получает на вход аудиофайл, загруженный пользователем. Внутри системы этот файл используется как напрямую, так и с извлечением из него текста, характеристик и спектрограмм. После применения модели машинного обучения система возвращает пользователю предсказание и вероятностное распределение для каждого из возможных настроений.

При обучении модели входными данными являются необработанные аудиоданные, спектрограммы, характеристики музыки, текст, а также целевые метки в строковом формате. Выходными данными являются числовые значения функции потерь, метрик качества, текущие шаг, эпоха, скорость обучения.

Все числовые нецелые данные и веса моделей хранятся в формате чисел с плавающей точкой (4 байта). Текст разбивается на элементарные единицы – токены и кодируется целыми числами.

Входящие аудиофайлы должны соответствовать форматам WAV или MP3. Ограничений на частоту дискретизации или глубину кодирования нет, так как алгоритм выполняет перекодирование в формат, использующийся при обучении моделей: 16 кГц, 16 бит.

Система ориентирована на европейскую музыку, поэтому при выборе набора данных большая часть композиций должна соответствовать именно этому кластеру музыки. При обучении не используется лицензированная музыка, если ее лицензией не разрешается загрузка и использование аудиофайлов.

### 1.3.4 Требования к программным интерфейсам

Модуль API модели предоставляет программный интерфейс серверному модулю согласно стилю REST (Representational State Transfer).

Серверный модуль предоставляет интерфейс клиентскому модулю и сторонним приложениям аналогично с использованием решения REST. С помощью интерфейса серверного модуля обеспечивается безопасное использование системы.

Модуль обучения модели программный интерфейс не предоставляет и используется отдельно.

### 1.3.5 Требования к протоколам обмена

Все модули, описанные ранее, взаимодействуют друг с другом по протоколу HTTP.

Для переадресации запросов и работы со статическими ресурсами используется инструмент nginx.

### 1.3.6 Требования к пользовательским интерфейсам

Пользовательский интерфейс предоставляется в виде веб-приложения. При проектировании используется адаптивная верстка.

В приложении используется темная нейтральная цветовая палитра. Размер текста и компонентов подбирается так, чтобы все элементы интерфейса были легко читаемыми.

### 1.3.7 Требования к быстродействию

Выполнение предсказания моделью на единичном примере с использованием соответствующей требованиям к аппаратным ресурсам инфраструктуры, не должно превышать 1 секунду. Пакет из 64 примеров должен обрабатываться не более чем за 5 секунд. Время предварительной обработки данных, формирования и отправки ответа клиенту не в пиковую нагрузку не должно превышать 10 секунд.

### 1.3.8 Требования к аппаратным ресурсам

Для использования обученной модели необходимо наличие GPU с размером графической памяти не менее 4 ГБ. Выполнение предсказаний на центральном процессоре возможно, но время ответа в таком случае увеличивается до 3-5 секунд, что не соответствует требованиям к быстродействию, описанным в разделе 1.2.7. Размер оперативной памяти должен быть не менее 8 ГБ, количество физических ядер процессора не менее 4-ех.

При обучении модели для размещения исходного набора данных потребуется жесткий диск размером не менее 256 ГБ. Для ускорения операций чтения рекомендуется использовать твердотельный накопитель. Оптимальный для обучения размер графической памяти составляет 6 ГБ, количество физических ядер процессора – 6-8, размер оперативной памяти – 16 ГБ.

### 1.3.9 Требования к безопасности и надежности

Поскольку система не хранит уязвимые данные, механизмы аутентификации и авторизации в программном обеспечении не

предусмотрены, но серверный модуль и модуль API модели ведут журнал событий для выявления возможных атак и нарушений.

В системе предусмотрены механизмы запрета отправки слишком больших файлов и ограничения частоты запросов, что защищает аппаратное и программное обеспечение от ошибок.

Модульная структура обеспечивает независимую работу программных компонентов, и отказ одного из модулей не ведет к отказу всей системы.

### 1.4 Сравнение с аналогами

В области классификации музыки по ее настроению мало аналогичных открытых проектов. Часть из них основана на классических алгоритмах машинного обучения, ввиду чего точность таких моделей мала или для их работы необходимы данные, которые невозможно получить из аудиофайлов. С другой стороны, существуют предварительно обученные модели машинного обучения, которые предназначены для работы с аудио. Но такие модели необходимо дополнительно обучать на задаче классификации музыки, то есть такие проекты нельзя назвать конечными продуктами в задаче определения эмоциональной окраски композиций. В таблице 1 представлен анализ конкурентов.

Таблица 1 – сравнение разработки с аналогами

| Аналог | Сильные стороны | Слабые стороны |
|---|---|---|
| Предварительно обученные модели, например Wav2Vec, HuBERT, M-CTC-T | Сложные модели, показывающие отличную точность в общих задачах | Невозможно использование в задаче классификации музыки без дополнительного обучения. Обучались не на музыке, а на речи |
| BerkinSerin/Music-Mood-Classification (GitHub) | Модель обучена на большом объеме данных | Модель применима только при использовании дополнительной информации – энергичности песен, тональностей, имен авторов, дат выпуска и так далее, что делает ее |

| | | |
|---|---|---|
| | | неприменимой для классификации необработанного аудио |
| RahulGaonkar/Music-Sentiment-Analysis (GitHub); cookiestroke/Music-Sentiment-Analysis (GitHub) | Легковесные модели | Для классификации используется текст, то есть модели не применимы для музыки без слов |

Таким образом, целесообразность собственной разработки заключается в следующем:

1. Открытость архитектуры, возможность загрузить и использовать модель в собственных целях;
2. Наличие сервера, клиентского приложения и API модели для проверки возможностей решения;
3. Использование в качестве входных данных аудиофайлов вместо набора заранее вычисленных характеристик (тональностей, жанров, энергичности, позитивности и так далее);
4. Высокая точность относительно классических моделей машинного обучения, за счет использования современных подходов к построению архитектуры и за счет анализа разнородных данных.

# 2 АНАЛИЗ ТЕХНОЛОГИЙ ДЛЯ РЕШЕНИЯ ЗАДАЧИ

## 2.1 Выбор платформы для реализации

В рамках работы на аппаратное обеспечение накладываются ограничения: для обучения глубоких нейронных сетей рекомендуется использование графического процессора, быстродействующего жесткого диска и процессора, достаточного объема оперативной памяти. Характеристики аппаратного обеспечения указаны в разделе 1.3.8 – «Требования к аппаратным ресурсам». Разрабатываемое программное обеспечение использует ядра CUDA – технологию параллельных вычислений на графических процессорах, принадлежащей компании NVIDIA. Версия программно-аппаратного обеспечения CUDA, использующегося в данной работе – 11.8. Параллельные вычисления ускоряют обучение глубоких моделей и являются стандартом в современной разработке.

Поскольку технология CUDA доступна только на операционных системах семейств Windows и Linux, при разработке и внедрении могут использоваться только эти платформы. Привязки к конкретной операционной системе из этих семейств нет, поскольку остальные использующиеся инструменты являются межплатформенными. При разработке и тестировании используется как Windows (Windows 11), так и Linux (Fedora 41).

Разрабатываемый код исполняется в изолированных окружениях conda. Поскольку на изолированную среду не может влиять стороннее программное обеспечение, такой подход обеспечивает стабильность и безопасность разработки, позволяет воспроизводить результаты исполнения программ.

Языком программирования для разработки был выбран python 3.10, так как он обеспечивает возможность использования проверенных библиотек для машинного обучения и анализа данных. При разработке клиентского приложения используется язык программирования Java Script.

## 2.2 Выбор средств разработки и тестирования

В качестве среды разработки был выбран VS Code. Этот инструмент упрощает написание программ, сборку проекта, запуск кода и его отладку. Интегрированная среда разработки VS Code является легковесной и позволяет вести разработку на выбранных языках программирования.

Для проверки качества обучаемых моделей используются библиотеки TensorBoard и Matplotlib. Первая позволяет сохранять метрики качества модели для их последующего изучения. Вторая обеспечивает визуализацию различных данных, например спектрограмм, звукового сигнала, матрицы ошибок, распределения исходных данных и так далее.

При разработке используется система контроля версий Git и веб-сервис для размещения кода GitHub. Такой подход обеспечивает резервирование кода, возможность возврата к стабильной версии проекта и является одним из стандартов в разработке.

Для проверки качества разрабатываемого API используется инструмент Postman. Эта платформа позволяет тестировать создаваемые интерфейсы до их введения в эксплуатацию, отправляя составленные запросы на необходимый сервер. Является легковесным, простым и быстрым инструментом, поддерживающим необходимый функционал.

При создании, обучении и тестировании моделей используется библиотека PyTorch 2.7.0. С ее помощью можно использовать как заранее подготовленные компоненты моделей, алгоритмов или функций, так и создавать собственные для узкоспециализированных задач. Библиотека обеспечивает высокую производительность и отлично подходит для обучения глубоких моделей. В отличие от своего конкурента – Keras (API над Tensorflow), PyTorch не ограничивает использование версий библиотеки при работе с ядрами CUDA на Windows и позволяет размещать тензоры в общей памяти графического процессора (что увеличивает объем памяти

GPU, но снижает производительность). Кроме того, PyTorch позволяет использовать библиотеку torchaudio для загрузки и аугментации аудиофайлов и спектрограмм. Также многие предварительно обученные модели, например «трансформеры» от Hugging Face, интегрируются с PyTorch, так как создавались при помощи этой библиотеки.

Для создания безопасного серверного приложения используется библиотека Django. Она позволяет вести разработку на языке программирования python, выполнять тонкую настройку политик безопасности, имеет встроенную административную панель и позволяет реализовать стиль построения интерфейса REST.

Для обертки обученной модели в API используется FastAPI – этот сервис позволяет создавать высокопроизводительный программный интерфейс и поддерживает асинхронные запросы. Поскольку API модели будет исполняться отдельно от серверного модуля, но находясь с ним внутри локальной сети, настройка политик безопасности с помощью FastAPI не потребуется, так как внешние запросы предварительно будет обрабатывать модуль на основе Django.

Для разработки клиентского приложения используется библиотека React, так как с ее помощью можно создавать динамический и отзывчивый интерфейс.

Поскольку серверный модуль с Django используется как API для программ и клиента, но обслуживание статических файлов (стилей, файлов HTML и прочее) с его использованием – неоптимальное решение, то для маршрутизации запросов между клиентом и сервером и для обслуживания статических файлов используется обратный прокси-сервер Nginx. Такое решение позволяет четко разделить логику модулей и оставляет возможность для дальнейшего расширения и улучшения производительности всего серверного приложения.

## 2.3 Выбор архитектур моделей машинного обучения

### 2.3.1 Многослойный персептрон по Румельхарту

Многослойный персептрон (MLP) – это сеть из искусственных нейронов, состоящая из нескольких слоев (входной слой, несколько скрытых слоев и выходной слой). Каждый искусственный нейрон в такой модели получает на вход взвешенную сумму выходов со всех нейронов предыдущего слоя.

Затем, к каждому такому нейрону применяется какая-либо нелинейная функция активации, например из ряда описанных функций в разделе 2.3.5. Ввиду этого становится возможным обучение модели методом обратного распространения ошибки. В векторном виде выход одного слоя можно записать так:

$$y = f_{act}(Wx)$$

где

$W \in \mathbb{R}^{m,n}$ – матрица весов;

$x \in \mathbb{R}^n$ – вектор входных данных;

$y \in \mathbb{R}^m$ – вектор выходных данных.

Данная модель часто используется как в отдельности, так и внутри других, более продвинутых архитектур. В частности, в данной работе такой слой часто используется для вычисления проекций данных.

### 2.3.2 Сверточные нейронные сети

Входные данные для моделей могут представлять собой необработанный звуковой сигнал или мел-спектрограммы на основе этого сигнала. В таком случае размеры входных данных по временной оси могут достигать десятки миллионов элементов для сырых аудиоданных и десятки тысяч для спектрограмм. Очевидной становится потребность в снижении размерности этих данных по временной оси для последующего анализа.

Одним из эффективных методов для этого являются сверточные нейронные сети [4, 5] – модели, выполняющие свертку исходных данных с помощью обучаемых фильтров (ядер) – относительно малых векторов или матриц, последовательно умножаемым скалярно на фрагменты входящих данных. В результате получается вектор или матрица меньшего размера.

Поскольку аудиоданные и спектрограммы представимы в виде одномерных одноканальных или многоканальных последовательностей, для уменьшения их размерности следует использовать одномерную свертку.

В таком случае фильтр (ядро) свертки является вектором и применяется к входящей последовательности:

$$y(t) = b + \sum_{i=0}^{k-1} w(i) \cdot x(t+i)$$

где

$y(t) \in \mathbb{R}$ – выходной элемент на шаге *t*;

$w \in \mathbb{R}^k$ – ядро свертки;

$b \in \mathbb{R}$ – смещение (не обязательно);

$k \in \mathbb{N}$ – длина (размер) ядра свертки.

Если входящие данные многоканальные, то после свертки все каналы суммируются. Также свертка может выполняться с шагом свертки, а входные данные могут дополняться нулями.

Очень часто число фильтров свертки бывает большим, что помогает находить разные шаблоны в данных и лучше их описывать.

В качестве дополнительного сжимающего инструмента используют слои объединения (пулинга от англ. pooling). Такие слои последовательно объединяют несколько подряд идущих элементов в один. Существуют такие реализации объединения, например – выбор максимального элемента:

$$y = \max_{i=0,\dots k-1} x(t+k)$$

После выполнения операций свертки и пулинга данные сжимаются и становятся более информативными. На этом этапе появляется возможность эффективного использования архитектур обработки последовательностей.

### 2.3.3 Рекуррентные нейронные сети

Классическим в глубоком обучении считается использование рекуррентных нейронных сетей (RNN – recurrent neural networks). Такие сети подают выход с некоторого скрытого слоя на вход этого же слоя с добавлением следующего элемента входной последовательности. Так слой обрабатывает весь входящий ряд, формируя на выходе один или множество векторов скрытых состояний.

В современной практике наиболее распространенными считаются две более совершенные версии классических RNN – LSTM (Long-Short term memory – долгая краткосрочная память) и GRU (Gated Recurrent Unit – управляемый рекуррентный блок): в этих архитектурах предусмотрены механизмы, улучшающие «память» модели. В LSTM между рекуррентными блоками передается не только скрытое состояние, но и вектор контекста *C* (долгосрочной информации). При этом на его вычисление тратятся дополнительные ресурсы и весовые коэффициенты, но повышается точность. В GRU модели используются те же вычисления, что и в LSTM, не считая вычисления вектора *C*. За счет этого GRU быстрее и легче, но может терять в точности.

В ходе работы будет использоваться GRU, так как в большинстве задач он не уступает LSTM по точности, однако превосходит в производительности.

Уравнения, лежащие в основе GRU, описываются следующим образом:

$$r_t = \sigma(W_r[x_t, h_{t-1}] + b_r)$$

$$z_t = \sigma(W_z[x_t, h_{t-1}] + b_z)$$

$$\hat{h}_t = tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t$$

где

$x_t \in \mathbb{R}^{D_x}$; $x_t \subset X \in \mathbb{R}^{T,D_x}$ – входные данные на итерации t;

$h_t \in \mathbb{R}^{D_h}$ – вектор скрытого состояния на итерации t;

$W_r, W_z \in \mathbb{R}^{D_h,D_h+D_x}$ – обучаемые матрицы весов обновляющего и сбрасывающего блоков соответственно, $r_t$ и $z_t$ – выходы этих блоков;

$W_{xh}, W_{hh} \in \mathbb{R}^{D_h,D_x}$ – обучаемые матрицы блока извлечения новой информации из векторов $x_t$ и $h_{t-1}$ соответственно, $\hat{h}_t$ – выход этого блока;

$b_r, b_z, b_h \in \mathbb{R}$ – смещения для описанных блоков;

операцией $[x_t, h_{t-1}]$ обозначена операция конкатенации векторов;

$\sigma, tanh$ – функции активации (сигмоидальная и гиперболический тангенс), описанные в разделе 2.3.5.

Полученный вектор скрытого состояния $h_T$ с последнего шага или вся последовательность векторов $h_t$ являются выходом GRU и представляют собой сжатую информацию о входных данных с учетом их временного характера.

Часто в задачах обработки последовательностей используют следующий подход: на каждой итерации вычисляются два вектора скрытых состояний. Первый получается путем прохода по входящей последовательности $X$ слева на право. Второй – наоборот, вычисляется, начиная с последних временных шагов. Так модель получает возможность учитывать двусторонние связи в исходных данных. Архитектуры с использованием описанного механизма называют двунаправленными.

### 2.3.4 Архитектура Transformer

Другим способом анализа последовательностей является использование архитектуры «трансформер» [6] (от англ. Transformer). Основным достижением такой архитектуры является возможность анализа

длинных последовательностей за счет использования механизма «внимания» (от англ. Attention) и работе с матрицами вместо последовательностей.

Вторая особенность ускоряет обработку и позволяет выполнять параллельные вычисления, например, на GPU. С другой стороны, для того, чтобы модель различала входящие матрицы как последовательные данные, необходимо прибегать к позиционному кодированию. Например, использовать принцип синусоидального кодирования [6]:

$$\hat{p}_t^i = \begin{cases} \sin(\omega_k t), & i = 2k \\ \cos(\omega_k t), & i = 2k \end{cases} \qquad \omega_k = \frac{1}{10000^{2k/d_x}}$$

где

$\hat{p}$ – матрица-добавка к входным данным;

t – позиция по временной оси;

k – позиция по оси кодирования признаков;

$d_x$ – глубина кодирования (размерность вектора признаков).

Такой подход позволяет с легкостью кодировать входные данные для их использования внутри «трансформера» и не зависит от размера контекста (длины последовательности).

Принцип внимания позволяет придавать некоторым элементам последовательности большую значимость (усиление, увеличение значений), а не важные элементы, наоборот, делать менее значимыми (ослаблять, уменьшать их значения). В «трансформере» существует блок внимания, выполняющий преобразование входящих данных.

Для этого используются несколько матриц:

- $Q \in \mathbb{R}^{N_Q, D_k}$ – матрица скрытых состояний («запросов» от англ. query);

- $W_k \in \mathbb{R}^{D_X, D_k}$ – матрица весов для отражения входных данных в пространство «ключей» (key);

- $W_v \in \mathbb{R}^{D_X, D_X}$ - матрица отражения входных данных в пространство «преобразований» (value).

Тогда входная матрица $X \in \mathbb{R}^{N_X, D_X}$, выходная матрица $Y \in \mathbb{R}^{N_Q, D_X}$, где N обозначает длину последовательности, а D – глубину кодирования одного элемента.

Алгоритм исчисления блока внимания следующий:

1. $K = XW_k$ – матрица «ключей»;
2. $V = XW_v$ – матрица «преобразований»;
3. $E = \frac{QK^T}{\sqrt{D_k}}$ – масштабированное скалярное произведение дает оценку важности элементов последовательности E, но эта оценка не является вероятностным распределением;
4. $A = softmax(E)$, где $softmax(\overrightarrow{x_i}) = \frac{e^{\overrightarrow{x_i}}}{\sum_j^{N_X} e^{\overrightarrow{x_j}}}$ - вероятностное распределение важности элементов последовательности;
5. $Y = AV$ – выходная матрица, как преобразованная матрица X с учетом важности своих элементов.

Другая распространенная запись вычислений:

$$Y = softmax\left(\frac{QK^T}{\sqrt{D_k}}\right)V$$

На практике часто используют блок «само-внимания» (от англ. «self-attention»). В таком блоке поиск подходящего Q выполняет сам блок внимания, поэтому в такой реализации вводится матрица $W_Q \in \mathbb{R}^{D_X, D_Q}$, которая умножается на входную матрицу.

Чтобы улучшить точность модели часто используют внимание с несколькими «головами». В таком случае используют множество экземпляров матриц $W_k$, $W_Q, W_v$, что дает разнообразные шаблоны внимания, так как каждая матрица будет обучаться по-своему ввиду начального распределения весов. Это позволяет обрабатывать последовательности с разных точек зрения, находить разные зависимости. На выходных матрицах

выполняют операции конкатенации в матрицу $Y \in \mathbb{R}^{N_k, D_v \cdot H}$ и умножения на матрицу $W_O \in \mathbb{R}^{D_v \cdot H, D_X}$, где $H$ – количество «голов внимания». Так получают выход модели $O = YW_O$ исходной размерности $(N_Q, D_X)$.

Выход блока внимания складывают с исходной матрицей X (такой подход называют skip-connection – соединение в обход). Это позволяет избегать затухания градиентов – проблемы, при которой градиенты ошибок в глубоких сетях уменьшаются при обратном распространении ближе к начальным слоям, и веса в них перестают изменяться. Затем на полученной матрице выполняют нормализацию вдоль одного элемента мини-пакета (с использованием обучаемых скаляров для сдвига данных).

Полученную матрицу подают на вход многослойного персептрона, что позволяет «сохранять факты» о данных. Выход снова складывают с входом и нормализуют.

На практике лучшие результаты дает другой порядок: нормализация выполняется перед блоком внимания и перед MLP, а не вместе со сложением, как в классической архитектуре [7].

Все описанные блоки объединяются и вместе составляют блок-кодировщик. Отличие кодировщика от декодирующего блока в том, что второй кроме своих основных входных данных получает еще и матрицы K, Q, V из кодировщика, что позволяет ему, учитывая контекст исходной последовательности, строить новое представление данных. Такой блок наиболее часто применяется в задачах синтеза новых данных из старых, например, при генерации или переводе.

Блоки кодировщиков и декодирующие блоки можно повторять многократно, что усложняет модель, но может давать улучшение результатов на некоторых данных.

### 2.3.5 Функции активации и функции потерь

Выходы многих ранее описанных блоков необходимо подавать на вход нелинейных функций, чтобы сделать возможным алгоритм обратного распространения ошибки и обучения модели. В качестве таких функций на практике часто используют сигмоидальную функцию активации, ReLU, гиперболический тангенс, рассмотренную ранее softmax и другие.

Сигмоидальная функция активации (рисунок 2.1) может также использоваться на выходном слое для задачи бинарной классификации, и служить для масштабирования данных между 0 и 1, она представляет собой зависимость:

$$S(x) = \frac{1}{1 + e^{-x}}$$

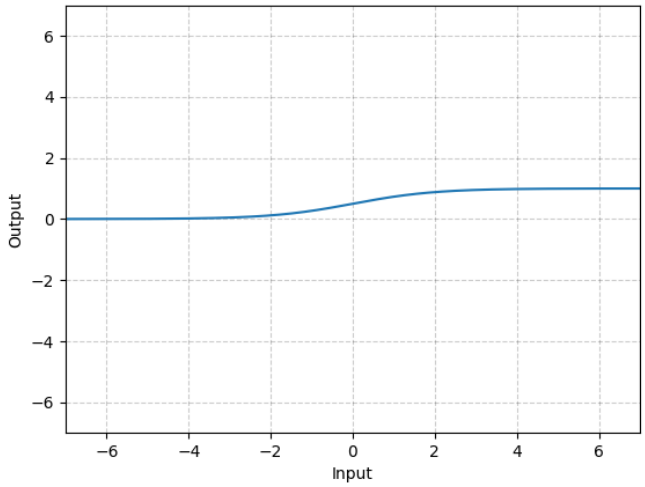



Рисунок 2.1 – Сигмоидальная функция активации

ReLU (Rectified Linear Unit) – выпрямленная линейная функция (рисунок 2.2), является одним из простейших примеров нелинейности и используется во многих скрытых слоях:

$$ReLU(x) = \max(0, x)$$

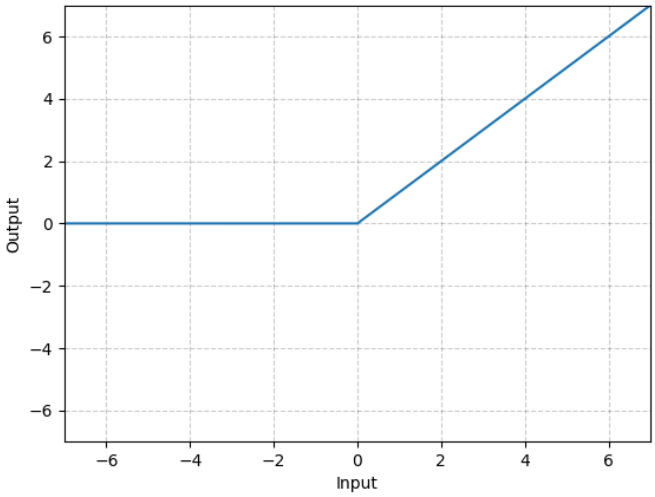



Рисунок 2.2 – Функция активации ReLU

В текущей работе чаще будет использоваться функция GELU (Gaussian Error Linear Unit) (рисунок 2.3), поскольку она предлагает плавный и гибкий способ активации нейронов, что оптимизирует градиентный спуск:

$$GELU(x) = \mathrm{x} \cdot \Phi$$

где

Ф – кумулятивная функция для распределения по Гауссу.

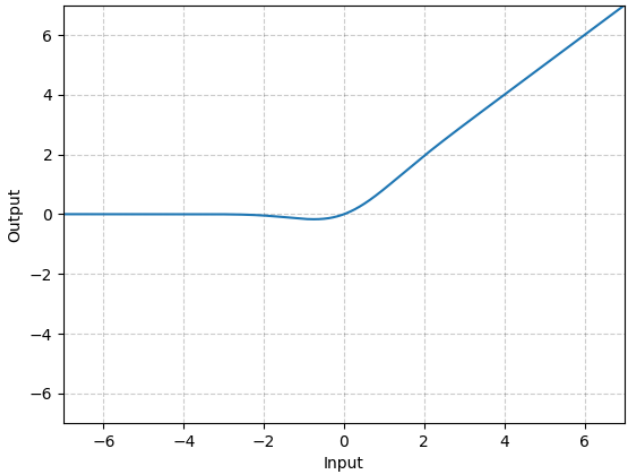



Рисунок 2.3 – Функция активации GELU

Гиперболический тангенс (рисунок 2.4) часто используется в рекуррентных нейронных сетях, масштабирует данные между -1 и 1, описывается формулой:

$$\sigma(x) = \frac{e^{x} - e^{-x}}{e^{-x} + e^{x}}$$

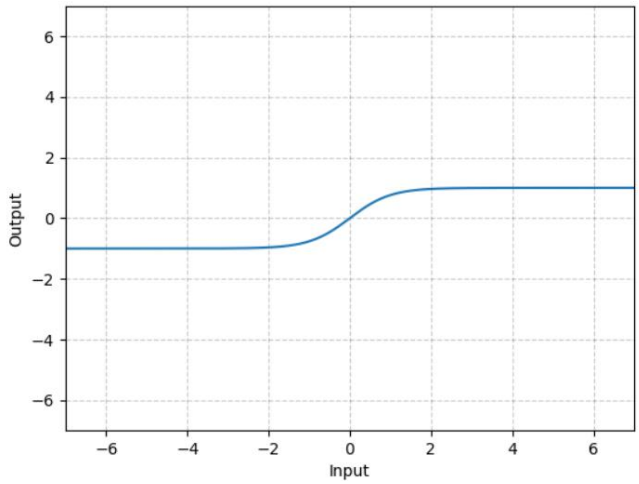



Рисунок 2.4 – Гиперболический тангенс

В качестве функции потерь при классификации используется категориальная перекрестная энтропия. Эта дифференцируемая функция позволяет определить ошибку модели при предсказании целевых классов. Каждому наблюдению в данных соответствует один из классов, устанавливая в векторе *y* единицу на соответствующем классу месте *i*. И, наоборот, классы, которые не соответствуют примеру, устанавливаются в 0. Тогда функция потерь вычисляется по формуле:

$$\mathcal{L} = -\sum_{i=0}^{n} y_i \cdot log\ \widehat{y}_i$$

Где

$y_i$ – ожидаемая вероятность класса *i* (1 или 0);

$\widehat{y}_i$ – предсказание модели или уверенность модели в классе *I*;

*n* – число классов.

С помощью этой функции определяется разница между ожидаемым и предсказанным вероятностными распределениями.

# 3 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ПРОДУКТА

## 3.1 Описание модульной структуры приложения

Разрабатываемое приложение разделяется на четыре программных модуля:

1. Модуль обучения модели. Не зависит от остальных модулей, поэтому разрабатывается, тестируется и исполняется отдельно. Решает задачи нереального масштаба времени: предварительная обработка данных, загрузка, обучение, тестирование и сохранение моделей. Разделяется на 7 подмодулей:

   a. Основной подмодуль (далее main). Является входной точкой программы и позволяет запускать необходимые подмодули, обрабатывает данные, передаваемые как аргументы с консольного интерфейса. С помощью этого модуля возможны запуск обучения модели заданного типа, продолжение обучения модели с заданной точки остановки и тестирование модели на отложенной выборке. Обеспечивает конфигурирование остальных подмодулей;

   b. Конфигурационный подмодуль (далее config). Хранит часто используемые константы, переменные окружения и прочие общие данные;

   c. Подмодуль обучения (далее train). Выполняет обучение и валидацию модели, сохраняет модели в локальное хранилище;

   d. Подмодуль архитектуры (далее model). Описывает модели и их компоненты, позволяет загружать их для последующего обучения;

   e. Подмодуль данных (далее data). Выполняет загрузку набора данных, их подготовку к обучению (объединение в мини-пакеты, нормализация, трансформирование размерностей) и их аугментацию;

f. Подмодуль статистики (далее stats). Выполняет задачу сбора статистики по исходным данным для их изучения. Исполняется как отдельный скрипт;

g. Подмодуль предобработки данных (далее preprocess). Выполняет предварительную обработку исходных данных. Очищает целевые метки, объединяет похожие настроения, сохраняет обработанный набор данных. Исполняется как отдельный скрипт, позволяет задать исходный набор данных, целевые возможные классов (настроения) и настроения, которые следует удалить из набора данных.

2. Модуль API модели. Модуль предназначен для запуска модели внутри изолированного окружения, оборачивает модель в интерфейс и обрабатывает запросы от других модулей. Подразумевает наличие хотя бы одной обученной модели. Использование этого модуля обусловлено необходимостью разделения логики и улучшения отказоустойчивости системы: сбой, связанный с использованием модели машинного обучения не повлечет за собой остановку серверного модуля, следовательно, клиентское приложение будет доступно при отказе модели. Состоит из двух подмодулей:

a. Модуль интерфейса (далее api). Обрабатывает запросы к модели, идущие локально на порт 5000. Подразумевает одну конечную точку: «/api/model/predict», получающей на вход HTTP POST запрос на предсказание модели model_ver по файлу audio;

b. Модуль инструментов (далее utils). Загружает и использует модель для построения предсказаний с использованием скользящего окна и усреднения вероятностей. Возвращает ответ модели и вероятностное распределение.

3. Серверный модуль. Модуль предназначен для обработки общих запросов клиентского приложения. Обрабатывает внешние запросы, перенаправляемы сервером nginx, слушает порт 8000. Подразумевает две

конечные точки: «/api/predict/file», «/api/predict/link». Первая обеспечивает предварительную обработку файла и его отправку на API модели. Вторая точка позволяет скачивать аудиофайлы по ссылке и аналогично передавать их на API модели. Является легко расширяемым модулем без зависимости от загруженной модели машинного обучения и позволяет настраивать политики безопасности. Состоит из двух подмодулей:

a. Подмодуль приложения (далее app). Определяет все разрешенные пути, использует обработчики запросов, проверяет корректность входящих данных;

b. Конфигурационный подмодуль (далее config). Определяет все необходимые политики безопасности, используемые библиотеки и последовательность их применения к входящему запросу.

4. Клиентский модуль. Модуль предназначен для обработки действий пользователя и отображения пользовательского интерфейса. Выполняет возможные проверки вводимых данных и отправляет запросы на сервер. Пользователь обладает возможностью загрузки аудиофайла как из локального хранилища, так и передачи серверу ссылки на сторонний ресурс Jamendo – каталог музыки с возможностью скачивания некоторых композиций (согласно лицензионным соглашениям).

Содержание и формат запросов и ответов сервера и клиента обозначен ниже:

1. Для конечной точки /api/predict/file тело POST запроса выглядит так:

```
{
  "file": <binary file>,
  "model_version": “1.0”
};
```

2. Для конечной точки /api/predict/link тело POST запроса выглядит так:

```
{
 "link": “http link”,
 "model_version": “1.0”
};
```

3. Ответ на оба описанных запроса представляет собой предсказание модели и вероятностное распределение классов:

```
{
 "label": "happy",
 "probabilities": {
  "happy": 0.82,
  "sad": 0.10,
  "relaxing": 0.05,
  "energetic": 0.03
 }
}.
```

Модульная структура приложения представлена на диаграмме пакетов (рисунок 3.1).

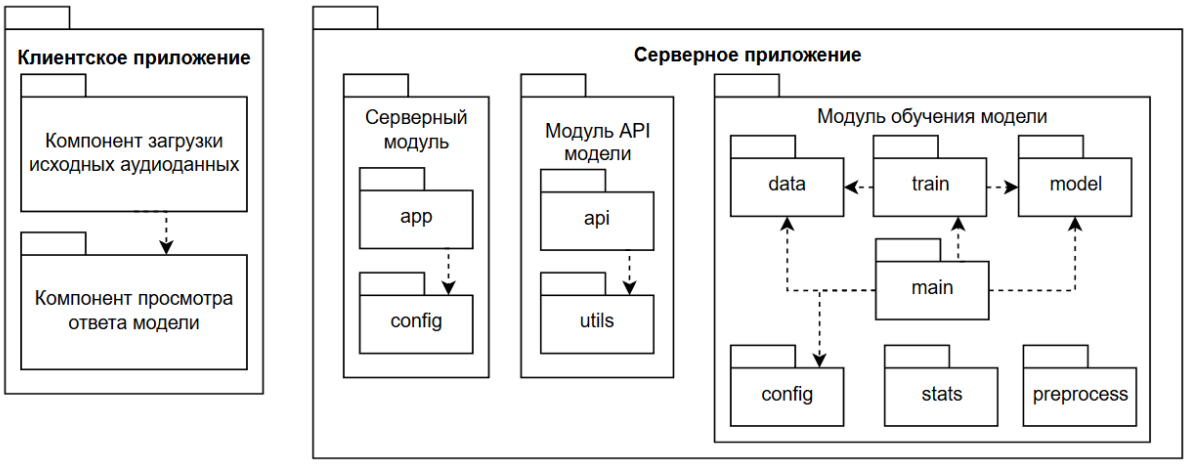



Рисунок 3.1 – Диаграмма пакетов

Характер входящих и исходящих данных, а также физическое размещение модулей представлено на диаграмме развертывания (рисунок 3.2).

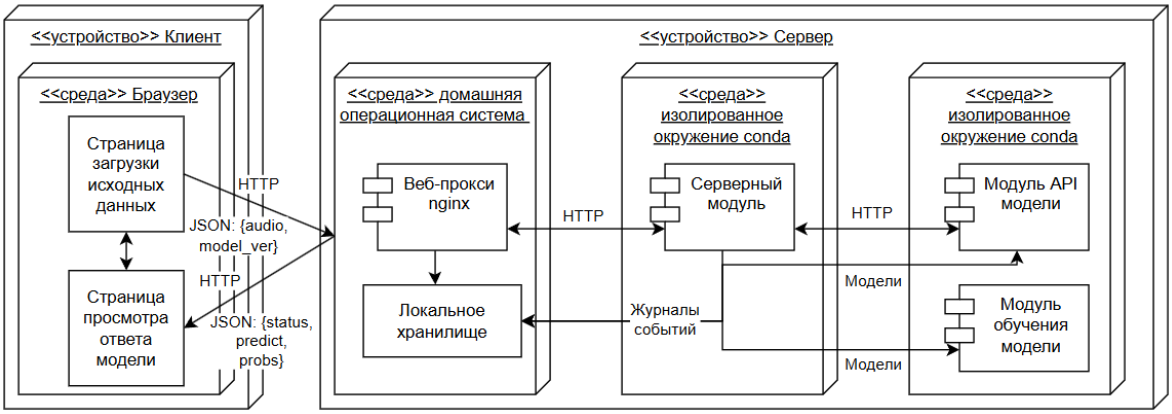



Рисунок 3.2 – Диаграмма развертывания

Используемые инструменты и интерфейсы представлены на диаграмме компонентов (рисунок 3.3).

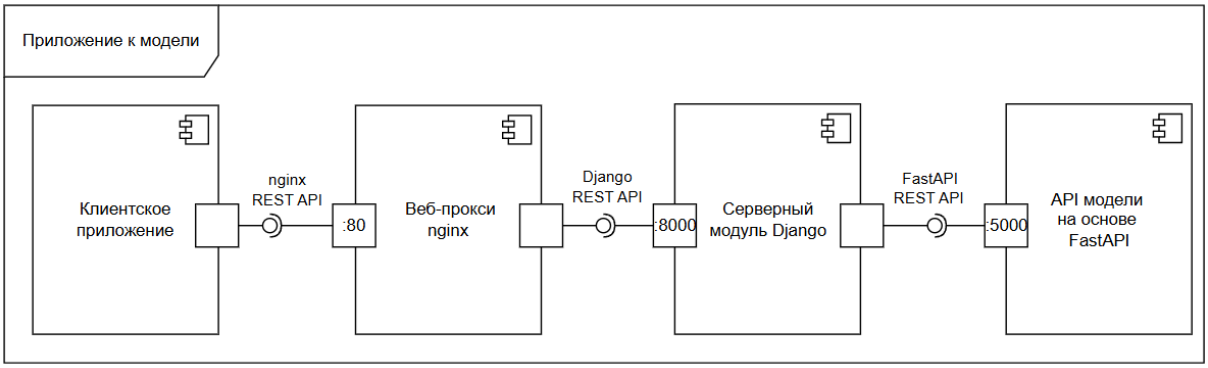



Рисунок 3.3 – Диаграмма компонентов

## 3.2 Описание данных

### 3.2.1 Общий обзор

В качестве основного набора данных используется открытый набор MTG/Jamendo v3.7 [8]. Набор данных содержит 18486 аудиозаписей (со свободной лицензией) в формате MP3, каждой из которых соответствует ряд меток, связанных с настроением или тематикой музыки. Распределение 40 самых часто встречаемых классов представлено на рисунке 3.4.

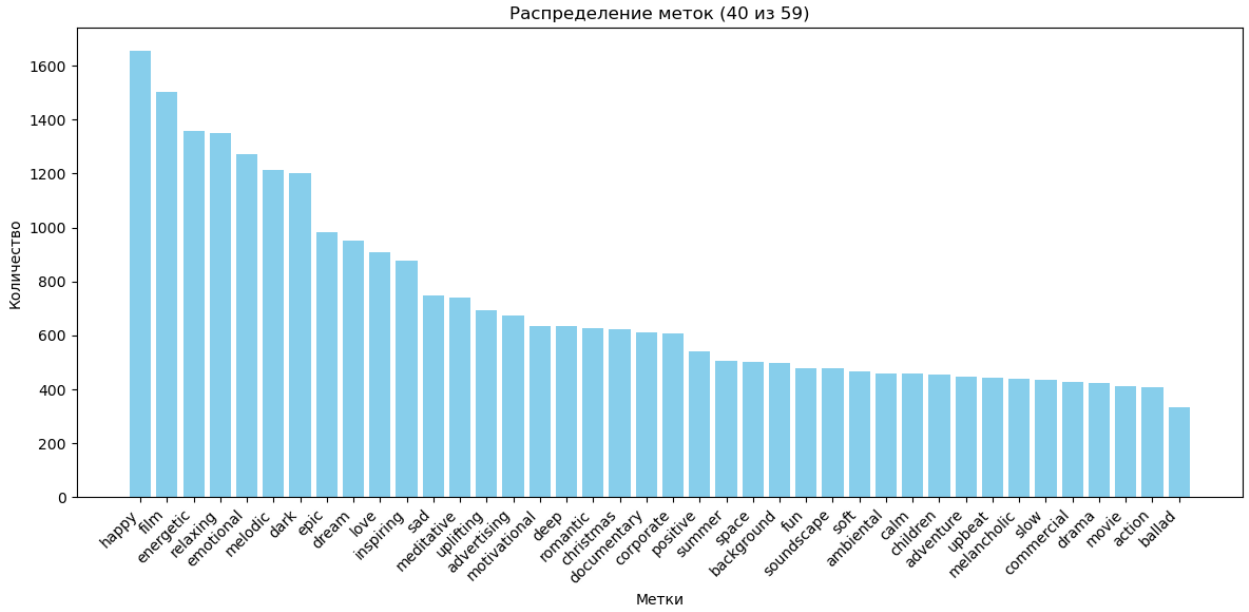



Рисунок 3.4 – Распределение меток в исходном наборе данных

Так как решаемая задача заключается в определении эмоциональной окраски, темы музыки («фильм», «приключение», «документальная») необходимо удалить или поменять на настроения. Также некоторые метки могут быть слишком узкоспециализированными («рождество», «детская»), поэтому их аналогично следует объединить с более важными метками или удалить. После применения алгоритма, описанного в разделе 3.4, были получены новые распределения, которые лучше подходят для обучения модели. Часть меток (и соответственно аудиозаписей) было необходимо убрать из выборки. В таком случае размер набора данных составил примерно 14 000 аудиозаписей.

```
Tags distribution after merging moods:
tags
happy        4177
relaxing     3715
sad          3543
energetic    2628
Name: count, dtype: int64
```

Рисунок 3.5 – Распределение меток после их слияния на 4-ех классах («веселый», «грустный», «энергичный», «спокойный», 14 064 элементов)

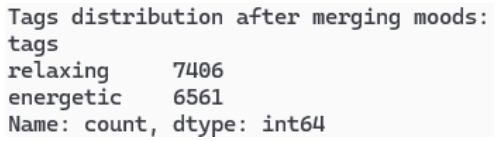



Рисунок 3.6 – Распределение меток после их слияния на 2-ух классах («энергичный», «спокойный», 13 967 элементов)


Рисунок 3.7 – Распределение меток после их слияния на 2-ух классах («веселый», «грустный», 14 063 элементов)

Метки представляют собой строковые категориальные данные и перед использованием в обучении кодируются векторами, где 1 на месте *i* означает принадлежность примера к классу на месте *i*. Соответствие классов и индексов в векторе определяется заранее.

Также в наборе представлены мел-спектрограммы с характеристиками, описанными в разделе 3.2.3.

### 3.2.2 Звуковой сигнал

Музыкальные произведения, записанные на любые цифровые устройства, представляют собой последовательность дискретных отсчетов амплитуды сигнала через равные промежутки времени. Частота дискретизации (в англ. – sample rate) определяет интервал между отсчетами. Согласно теореме Котельникова для точного восстановления непрерывного сигнала по его дискретным отсчетам во времени частота дискретизации должна быть не менее чем в два раза больше, чем максимальная частота сигнала [9]. Чаще всего на практике для хранения музыкальных данных используют частоту дискретизации в 44100 Гц. Для задач машинного обучения такое качество не обязательно, поэтому частоту дискретизации берут равной 22050 Гц, 16000 Гц или 11025 Гц. Вариант в 16000 Гц является

оптимальным выбором и использовался во многих исследованиях и моделях, например, в Wav2Vec [10].

Глубину кодирования аудио часто приравнивают 16 битам. Ввиду этого, стандартного формата с плавающей точкой хватит для хранения нормализованных амплитуд сигнала (нормализация данных улучшает процесс обучения моделей).

Таким образом, данные, которые будут использоваться в работе, представляют собой нормализованные дискретные амплитуды сигналов с частотой дискретизации 16000 Гц и глубиной кодирования 16. Общее число отсчетов сигнала должно быть достаточным для покрытия нескольких секунд аудио, то есть достаточно взять от 32 до 128 тысяч отсчетов, для покрытия от 2 до 8 секунд музыкальных произведений. Выбор такой продолжительности также позволит выполнить аугментацию данных за счет случайного выбора последовательностей внутри одного и того же файла.

Для улучшения обобщающей способности моделей используется аугментация данных. В случае со звуковым сигналом к нему применяется:

- случайное затухание сигнала длинной в 10% аудио;
- случайный сдвиг аудио на 4 полутона вниз или вверх;
- случайное усиление сигнала в [0,6; 1,1] раз;
- применение случайного шума с максимальной амплитудой в 0,001.

В цикле обучения все аугментации, кроме наложения шума, выполняются с заданной вероятностью (обычно, p=0,3). Пример результата аугментации изображен на рисунке 3.8.

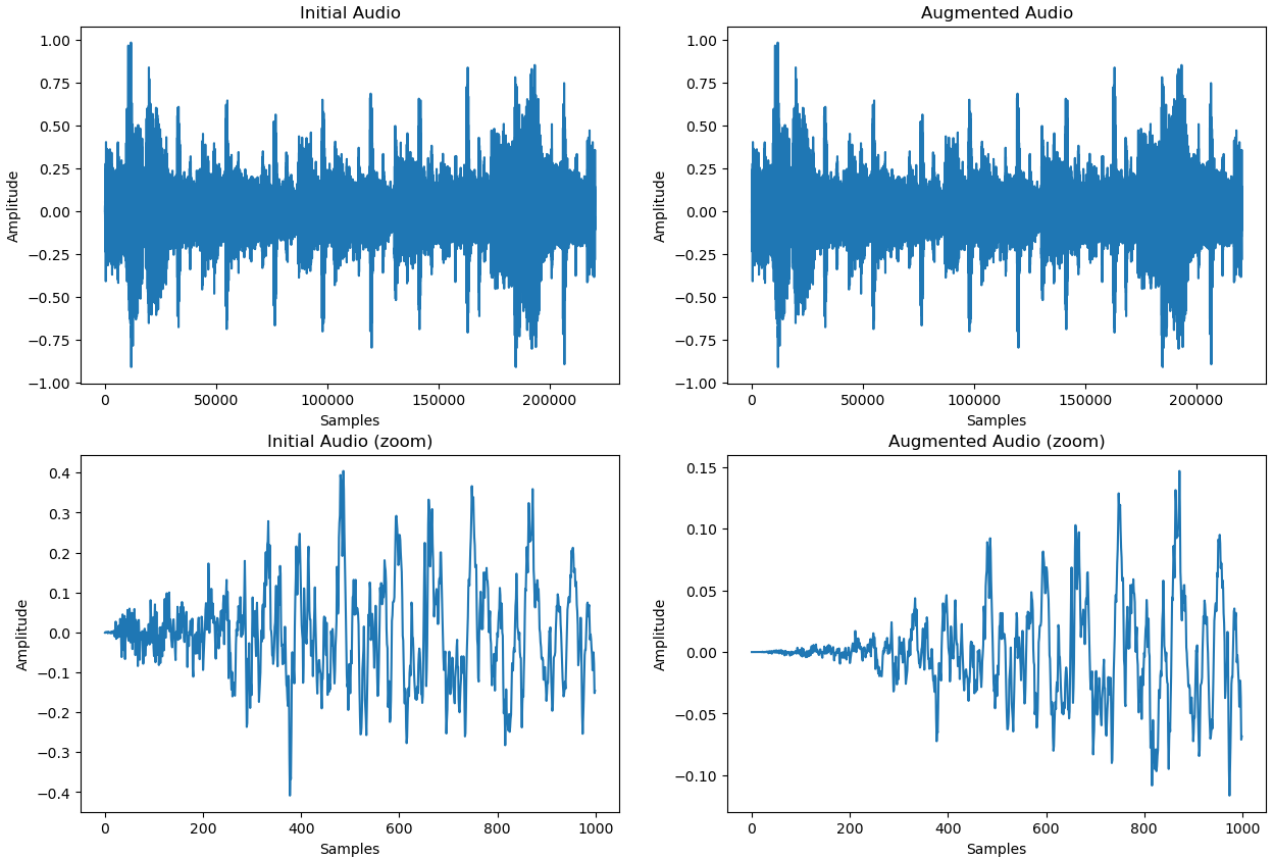



Рисунок 3.8 – Результат аугментации звукового сигнала

### 3.2.3 Мел-спектрограммы

Необработанные аудиоданные могут отражать не все характеристики музыки. Также ввиду их большого размера возможным становится покрытие всего нескольких секунд произведения. Поэтому при работе с аудио часто используют спектрограммы – представление исходного сигнала в виде изменения его частотного состава во времени. Спектрограммы строятся на основании преобразования Фурье на коротких отрезках исходного сигнала [11]. Такие отрезки называются окнами (или рамками, от англ. frame). Спектрограммы более удобные в обработке и могут отражать спектральные характеристики аудио.

Мел-спектрограмма – это отражение частотной оси обычных спектрограмм на логарифмическую ось. Такой подход не только позволяет сократить размерность данных, но и не вредит точности, так как такое

отражение повторяет особенности слуховой системы человека. В наборе данных используется отражение исходного спектра в 96 делений (мел), размер окна равен 512, а шаг - 256, используется окно Ханна. При построении спектрограмм разработчики набора данных использовали частоту дискретизации в 22050 Гц. Амплитуды в мел-спектрограммах записывались в децибелах (между -90 дБ и +27 дБ).

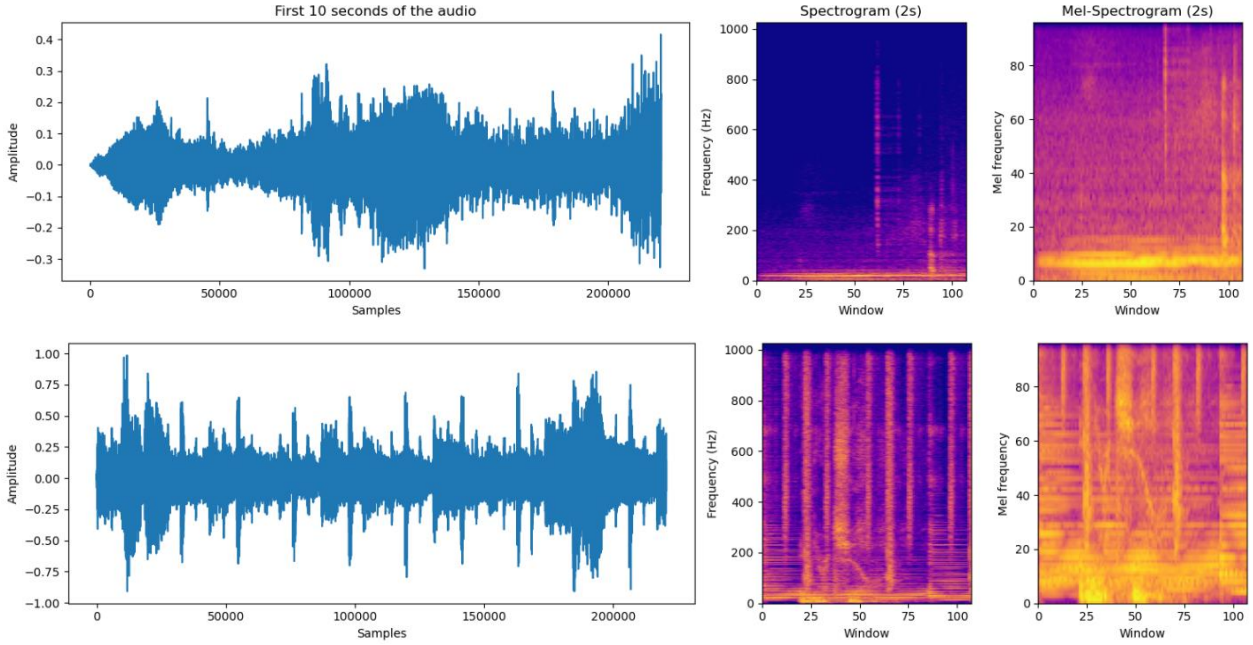



Рисунок 3.9 – Вид спектрограмм и исходного сигнала (10 секунд аудио и 2 секунды спектрограмм)

На вход модели следует подавать 5000-10000 окон, что позволит покрыть от 60 до 120 секунд исходного аудио. Также хорошим решением будет масштабирование мел-спектрограмм в диапазон чисел с плавающей точкой между 0 и 1.

Для аугментации спектрограмм используются следующие операции:

- маскирование случайного спектра частот. В работе маскированные участки заполняются в минимальный уровень сигнала (-90 дБ) для 6-ти случайных мел-делений;

- маскирование случайного временного отрезка. В работе маскируются случайные 64 временных деления (окна);

- случайное усиление или ослабление сигнала. К спектрограмме добавляется случайное число в промежутке от -5 до +5;

- применение случайного шума. К спектрограмме добавляется шум согласно равномерному распределению с математическим ожиданием 0 и стандартным отклонением 0,4.

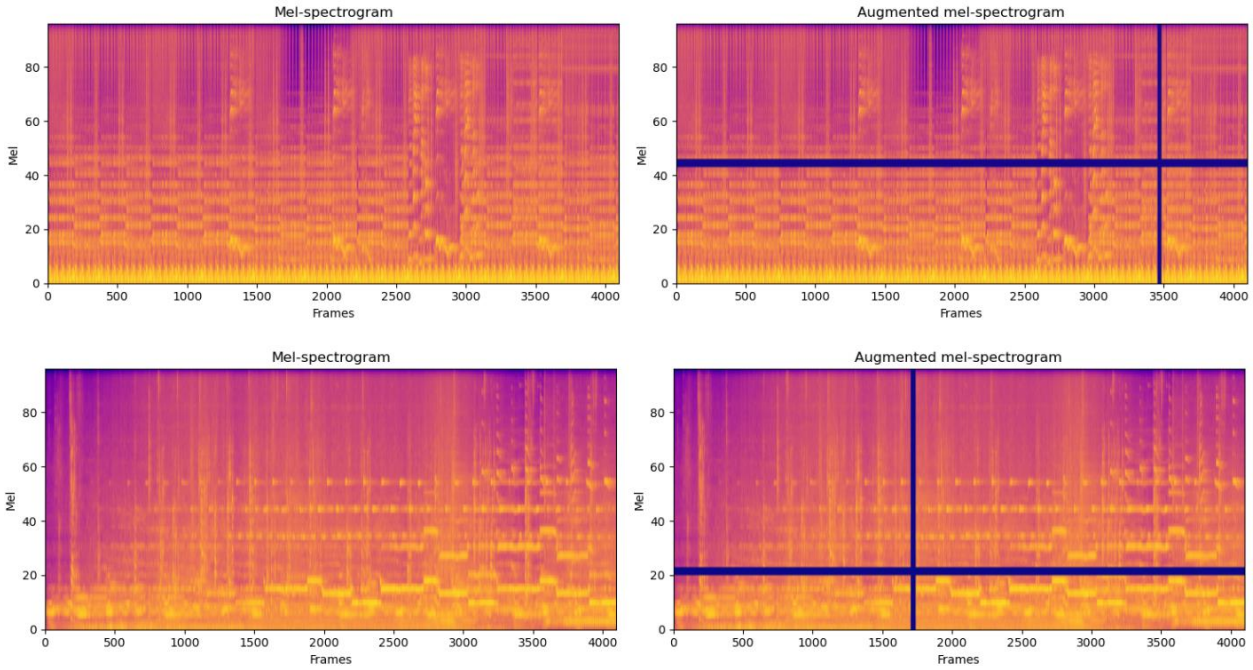



Рисунок 3.10 – Результат аугментации мел-спектрограммы

### 3.2.4 Текст

Для извлечения текста из звукового сигнала можно использовать предварительно обученные модели, например Whisper. Эта модель преобразует исходный сигнал в мел-спектрограмму, применяет к ней несколько слоев одномерной свертки и выполняет перевод звука в текст с помощью архитектуры трансформер. Выходом модели является последовательность целочисленных токенов. В Whisper токены кодируют не слово целиком, а его составные части, например корни, окончания, приставки. Так словарь Whisper размером чуть больше 50000 элементов покрывает 99 языков, включая русский [12].

Поскольку большая часть композиций в исходном наборе данных записана без слов, хорошим решением будет предварительно обучить классификатор на специализированном наборе данных.

Для этого был выбран набор данных XED [13], так как в нем представлено множество европейских языков, включая русский. Каждому предложению из набора соответствует одна из 8 меток, однако для решаемой задачи понадобятся всего две: «joy» и «sadness». Также каждое предложение следует закодировать с помощью токенизатора Whisper, чтобы модель была пригодна для использования в паре с самим Whisper.

Так, обработанный набор данных XED состоит из 36 608 предложений.

### 3.2.5 Характеристики сигнала

В качестве дополнительных данных могут использоваться характеристики звукового сигнала, которые описывают аудиофайл целиком. Для их извлечения исходный сигнал делится на окна размером в 512 дискретных отсчетов, затем для каждого временного интервала вычисляются:

1. Скорость пересечения нуля (Zero Crossing Rate) – число изменений знака амплитуды сигнала относительно общего числа дискретных отсчетов в заданном временном интервале;
2. Корень из среднего квадрата амплитуды сигнала (Root-Mean-Square Energy) – показатель локальной интенсивности сигнала;
3. Центр тяжести спектра (Spectral Centroid) – математически ожидаемая частота спектра (взвешенная сумма частотных делений, где веса – амплитуды соответствующих им частот);
4. Ширина спектра (Spectral Bandwidth) – стандартное отклонение спектрального распределения. Отражает степень рассеивания мощности вокруг центра тяжести спектра;
5. Граница частоты (Spectral Rolloff) – частота, ниже которой сосредоточено 85% спектральной энергии;

6. Плоскость спектра (Spectral Flatness) – отношение геометрического среднего спектральных компонент к их арифметическому среднему. Показывает, насколько спектр близок к белому шуму (плоский спектр, значения характеристики близкие к 1) или, наоборот, как много в нем присутствует тональных составляющих (выраженные пиковые частоты, значения характеристики стремятся к 0);

7. Мера изменения спектра (Spectral Flux) – сумма квадратов разностей одинаковых спектральных компонент между соседними временными интервалами. Отражает изменчивость сигнала, то есть наличие внезапных переходов, ритмических ударов, степень атаки сигнала;

8. Хроматические признаки (Chroma Features) – 12 характеристик, каждая из которых соответствует энергии двенадцати полутоновым классам (нотам) вне зависимости от их октавы (то есть накопленная энергия для 12 нот с чередованием по всей спектрограмме). Позволяет определить тональность, аккорды, мелодические последовательности;

9. Аккордово-тональные характеристик (Tonnetz Features) – набор из 6 признаков, отражающих зависимости между хроматическими характеристиками. Отражает расстояния между аккордами и мелодическими структурами.

Получаемые характеристики усредняются для всех временных окон, также находятся их стандартное отклонение, 25, 50 и 75 перцентили. Такой подход позволяет описать целое музыкальное произведение в виде вектора из 125 характеристик (25 признаков по 5 статистик для каждого).

Также извлекаются признаки, не являющиеся изменчивыми:

1. Темп (tempo, BPM - beats per minute) - число регулярных периодических атак (заметных скачков амплитуды сигнала) в минуту;

2. Частота появления атак (Onset Rate) - общее число атак в секунду (не только периодических).

Характеристики масштабируются до диапазона чисел с плавающей точкой между 0 и 1.

### 3.3 Описание алгоритма предварительной обработки набора данных

Набор данных предварительно обрабатывается для очистки и улучшения процесса обучения (рисунок 3.11).

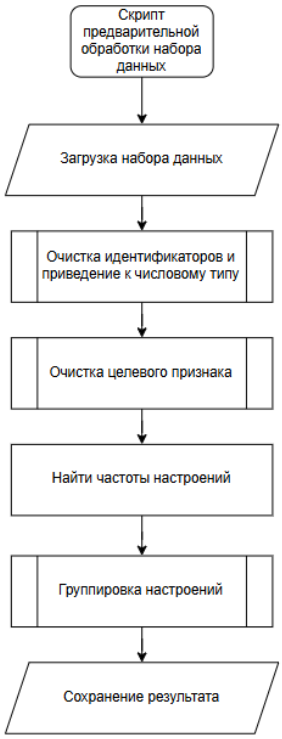



Рисунок 3.11 – алгоритм предварительной обработки

Целевые метки – настроения, в исходном наборе данных часто являются пересекающими, например класс «sad» очень схож с тегом «melancholic», а «energetic» - похож на «epic». Кроме того, в наборе представлены не только настроения, но и темы произведений. Например, метки «commercial», «summer», «nature» могут ухудшать сходимость обучения.

Ввиду этого был разработан алгоритм, который объединяет похожие настроения на основании словаря с частотами их совместного присваивания одному примеру, а также алгоритм убирает запрещенные настроения и темы,

оставляя только целевые классы. Блок-схема алгоритма представлена в приложении А.

### 3.4 Описание архитектуры модели машинного обучения

Поскольку входные данные являются длинными временными последовательностями, перед их анализом следует выполнить уменьшение размера по временной оси. Для этого подходит одномерная сверточная сеть. Такой подход часто используется в задачах работы с аудио [10, 14, 15].

После свертки модели следует выполнить перестановку осей: временную ось и ось кодирования (ось каналов, признаков) следует поменять местами (для последующей обработки в более сложных слоях).

Скрытое малоразмерное представление последовательностей можно использовать в таких архитектурах, как рекуррентные нейронные сети или трансформер. Можно совместить оба подхода или выбрать один из них. Рекуррентные нейронные сети лучше создавать двунаправленными, а в трансформере использовать позиционное кодирование и множество голов внимания. Возможно, на данном уровне понадобится проекция данных по одной оси для совместимости обучаемых весовых матриц и входных данных. Для такой задачи подойдет MLP с функцией активации ReLU.

Выход трансформера и рекуррентных нейронных сетей представляет собой трехмерное скрытое представление данных. На этом этапе необходимо преобразовать данные из трехмерных в двумерные. Для этого можно использовать блок внимания (желательно, с множеством голов). При этом следует ввести обучаемый вектор $Q \in \mathbb{R}^{1,D_k}$, в таком случае выход блока внимания будет векторным: $Y \in \mathbb{R}^{1,D_X}$. Такой подход эквивалентен усреднению скрытого представления по временной оси с учетом важности отдельных элементов исходной последовательности.

Полученное скрытое состояние представляет собой сжатое описание исходных данных, которое можно преобразовать в вероятностное

распределение. Для этого следует использовать MLP с функцией активации softmax на выходном слое.

Таким образом, архитектура может быть представлена как конвейер CNN → GRU → Transformer. С учетом необходимых дополнительных слоев, архитектура изображена на рисунке 3.12.

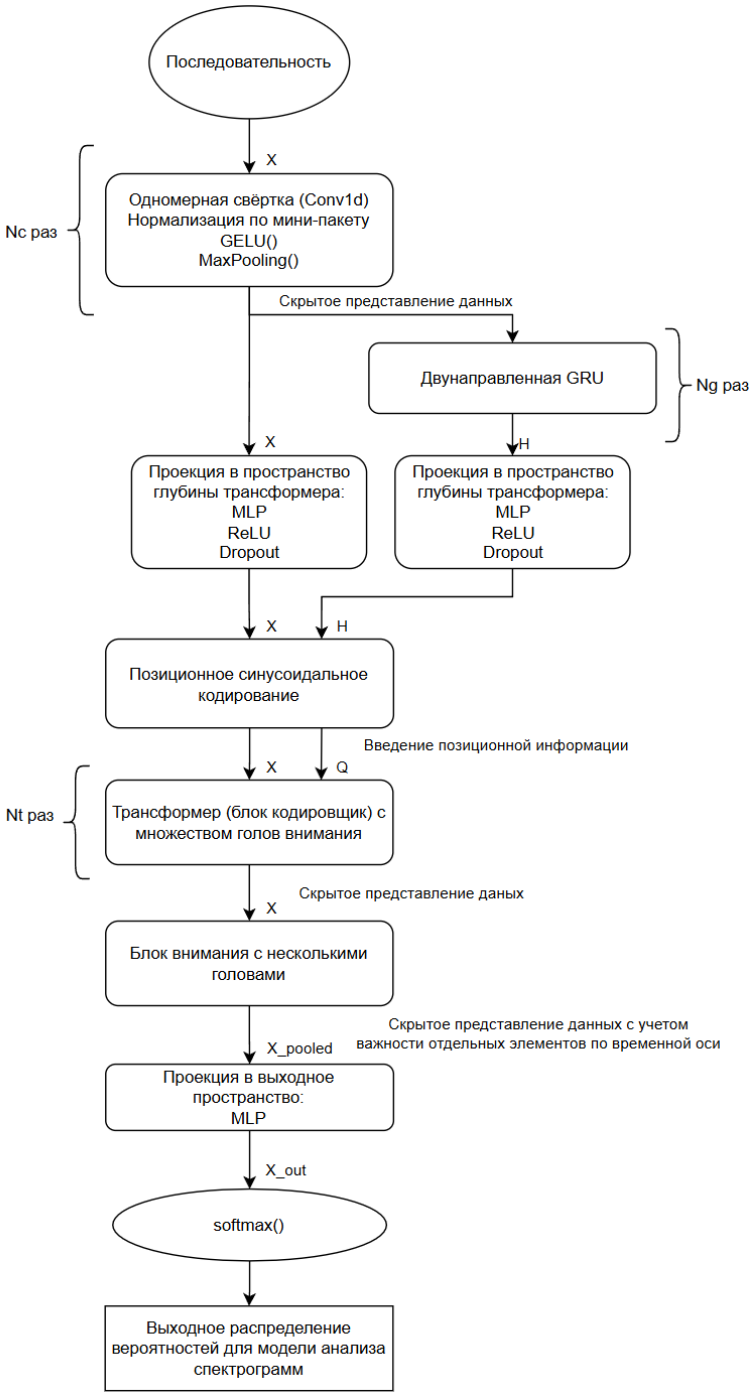



Рисунок 3.12 – базовая архитектура для анализа последовательностей

С использованием подобной архитектуры становится возможным анализ длинных временных последовательностей.

Для проверки возможностей различных архитектур при одинаковых параметрах были запущены обучающие циклы. В результате сравнения было установлено, что описанная выше архитектура является наилучшей в решаемой задаче. Значения метрик качества и функции потерь при обучении архитектуры CNN → GRU (фиолетовый график на рисунке 3.12) показали худшую сходимость, чем при обучении CNN → GRU → Transformer (синий график на рисунке 3.13).

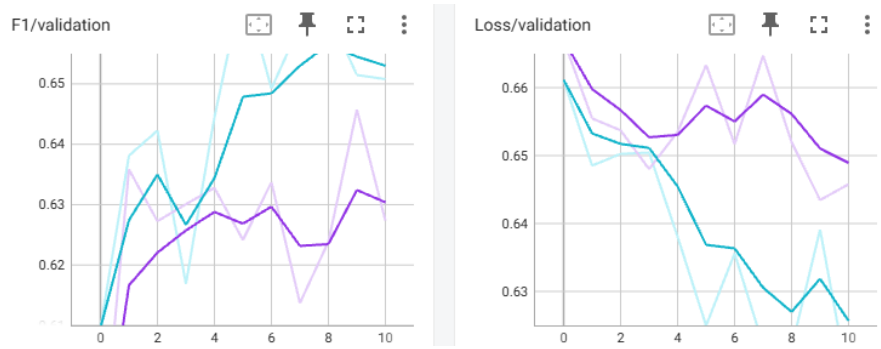



Рисунок 3.13 – Сравнение метрики $F_1$ и функции потерь для архитектур с трансформером (синий) и без (фиолетовый)

Применение архитектуры CNN → Transformer (зеленый график на рисунке 3.14) показало более качественные результаты, чем модель CNN → GRU, однако архитектура с применением GRU для поиска векторов запроса Q для трансформера снова оказалась лучше.

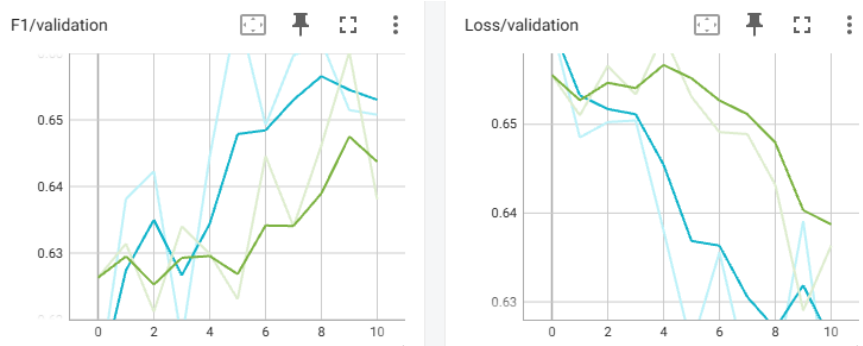



Рисунок 3.14 – Сравнение метрики $F_1$ и функции потерь для архитектур с GRU (синий) и без (зеленый)

Ввиду этого для обработки спектрограмм и звукового сигнала используется конвейер CNN → GRU → Transformer.

Для определения эмоциональной окраски текста наличие сверточной нейронной сети не обязательно. Для такого источника данных достаточно двух слоев: слоя, преобразующего исходные токены в сжатое векторное представление (слой эмбеддинга, от англ. Embedding) и трансформера. Схема подобной архитектуры изображена на рисунке 3.15.

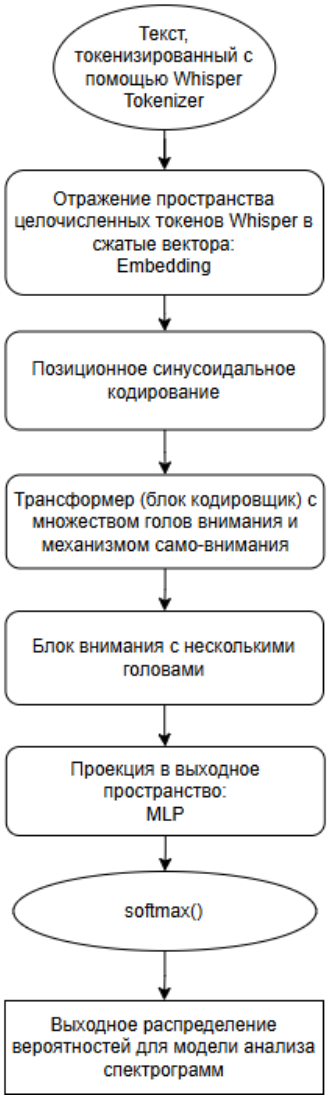



Рисунок 3.15 – Архитектура модели для анализа эмоциональной окраски текста

Объединение нескольких моделей для анализа гетерогенных данных позволит улучшить точность предсказаний. Для этого достаточно выходы

трех предварительно обученных моделей перед применением проекций и отражением в вероятностное пространство конкатенировать и подать на вход общего блока внимания. При этом для композиций, где нет слов, часть модели выключается из работы.

В результате будет получено двумерное сжатое представление данных с учетом важности выводов каждой из моделей.

Подобная архитектура, вместе с конвейером обработки данных, представлена на рисунке 3.16.

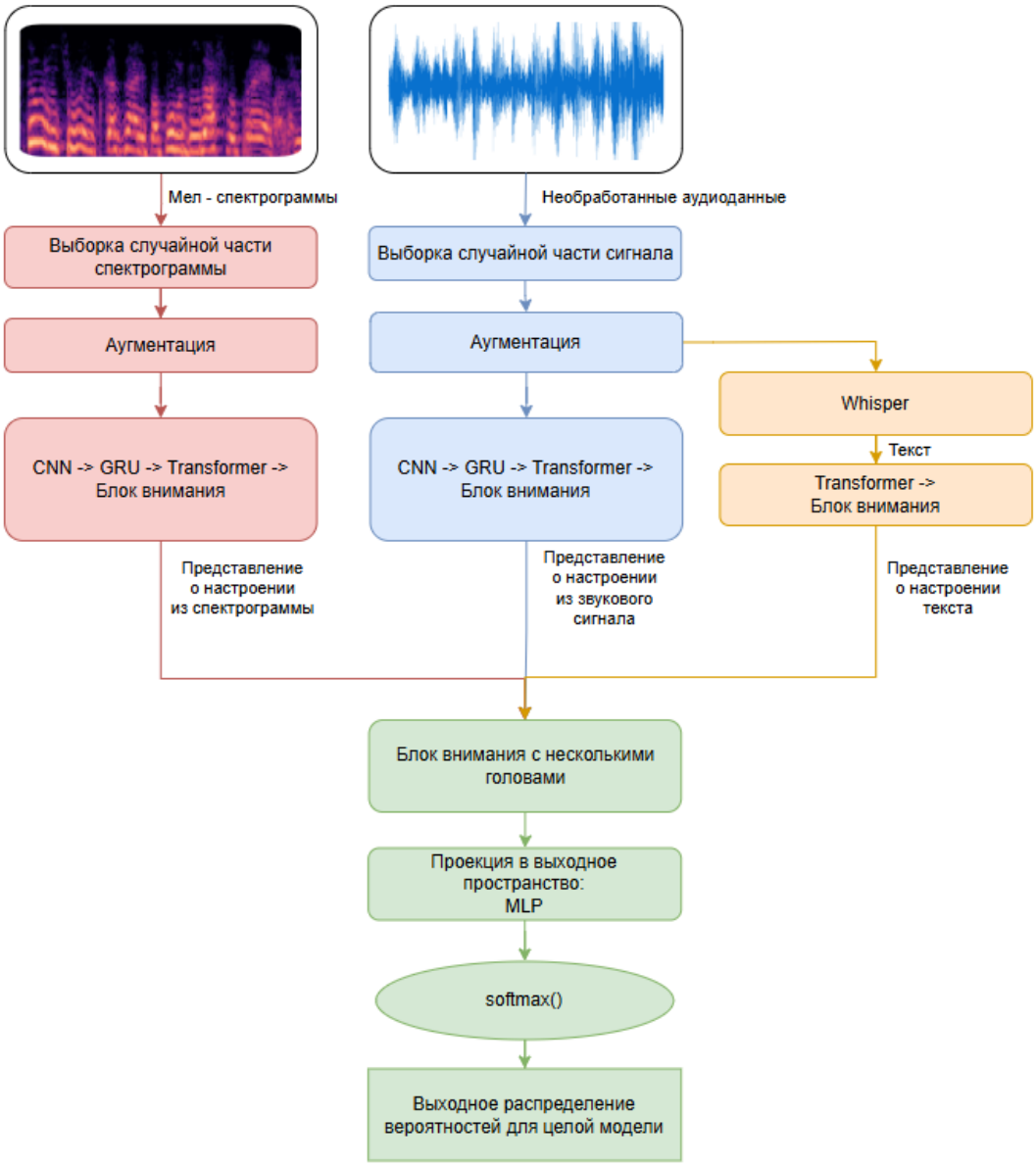



Рисунок 3.16 – конвейер обработки и анализа разнородных данных

### 3.5 Обучение модели анализа спектрограмм

Перед обучением данные разделялись на тренировочную и тестовую выборки в соотношении 80% на 20%. Тренировочные данные разделялись на 5 выборок, на каждой итерации для обучения выбирались 4 выборки, 5-я использовалась для валидации модели. Такой подход называют перекрестной валидацией (k-fold).

В качестве метрик качества были выбраны метрики Precision, Recall и $F_1$Score. При обучении модели на задаче классификации четырех меток («happy», «sad», «relax», «energetic») был выявлен эффект переобучения модели. Несмотря на высокие значения метрик на тренировочной выборке (около 0,8), на тестовых и валидационных данных модель ошибалась часто – значения метрик качества составили приблизительно 0,6 (рисунки 3.17 и 3.18).

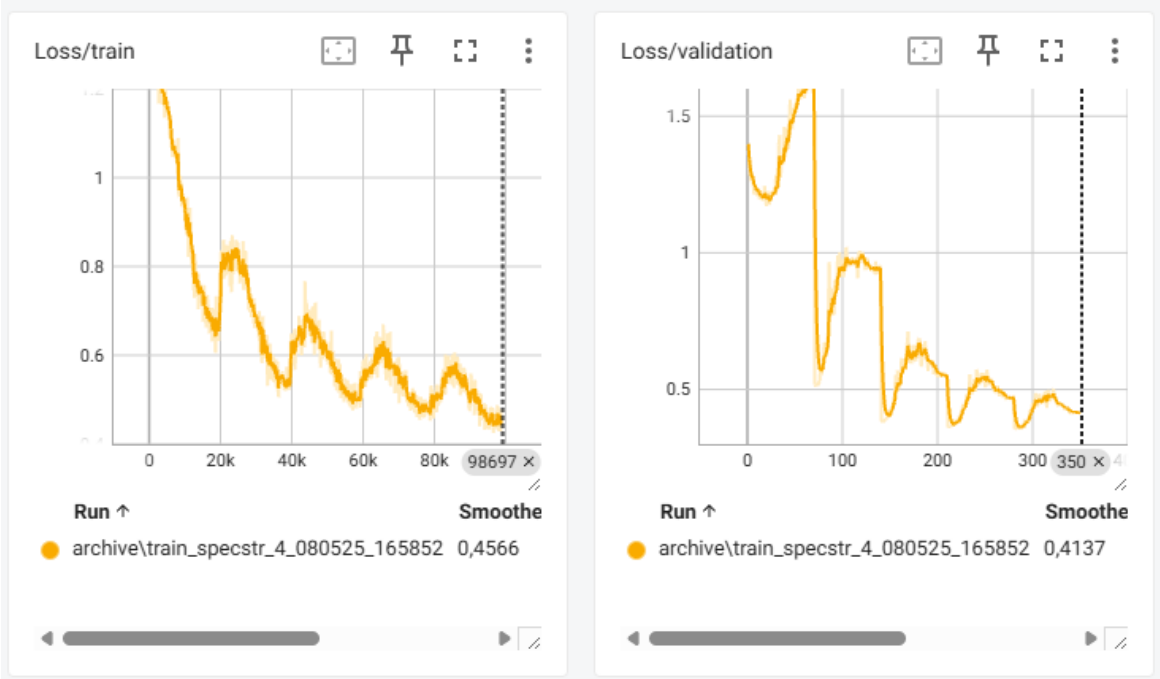



Рисунок 3.17 – Процесс обучения модели анализа спектрограмм на 4-ех классах с 5-ю разделениями выборки в течение 70 эпох (350 итераций)

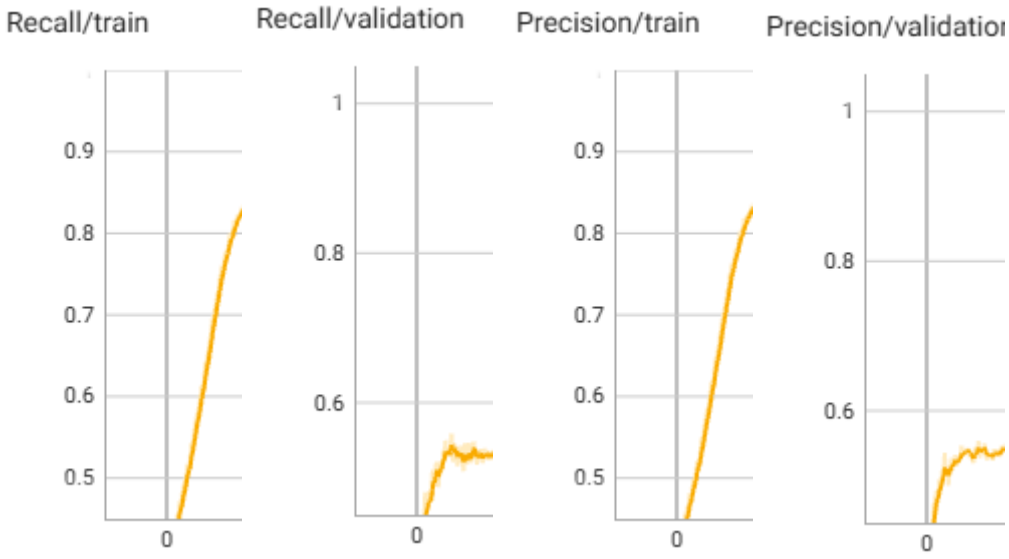



Рисунок 3.18 – Эффект переобучения

Для преодоления этого эффекта были введены следующие механизмы:

1. Dropout – случайное «выключение» весов модели при обучении путем установки в 0 части выходных данных с предыдущего слоя (обычно маскировались 10-40% выходов);

2. Регуляризация L2 – введение в функцию потерь штрафа в размере суммы квадратов весов модели, умноженных на коэффициент регуляризации (использовались коэффициенты в пределах 1e-2/1e-4);

3. Ранняя остановка цикла обучения, если функция потерь на валидационных данных не уменьшалась несколько эпох (обычно в течение 5-7 эпох);

4. Увеличена вероятность случайных аугментаций.

Для улучшения сходимости модели и качества обучения были введены планировщики скорости обучения (пример графиков скорости обучения представлен на рисунке 3.19):

1. OneCycleLR – увеличение скорости обучения в 10 раз первые 30% эпох с ее последующим уменьшением до 1e-6;

2. ReduceLROnPlateau – уменьшение скорости в 10 раз, если функция потерь на валидационных данных не уменьшалась 3-5 эпох.

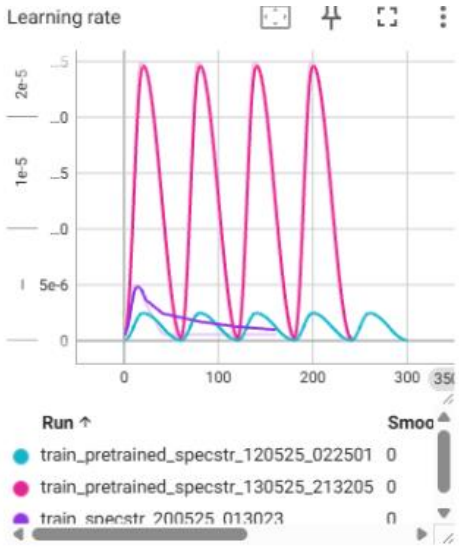



Рисунок 3.19 – Графики изменения скорости обучения с использованием планировщиков

В результате проблема переобучения была решена и обобщающая способность модели повысилась. Ввиду этого значения метрик и функции потерь сохраняются и на отложенных выборках (рисунок 3.20).

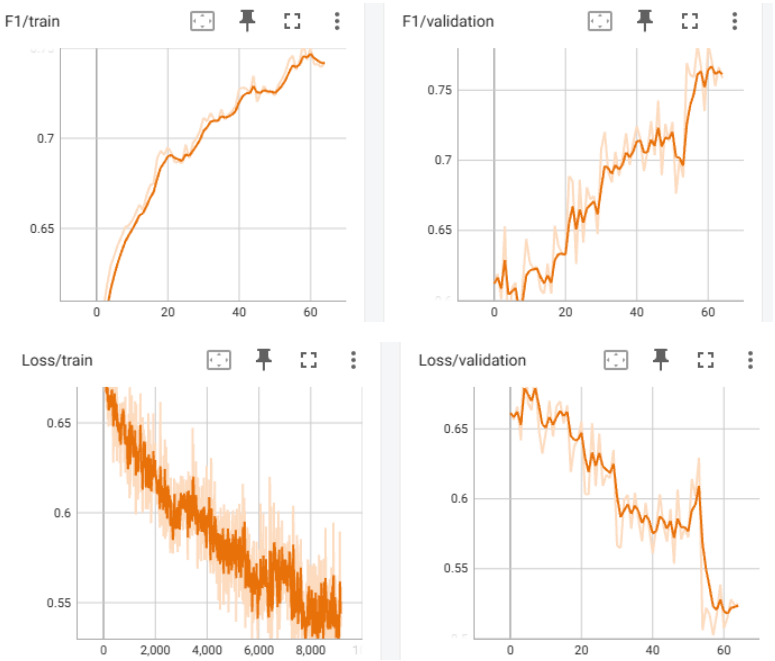



Рисунок 3.20 – Графики изменения значений метрики $F_1$ и функции потерь при обучении модели после введния новых механизмов

В результате тестирования и анализа матрицы ошибок при оценке модели была выявлена проблема: модель плохо различает классы «sad», «relaxing», а также часто ошибочно предсказывает класс «energetic».

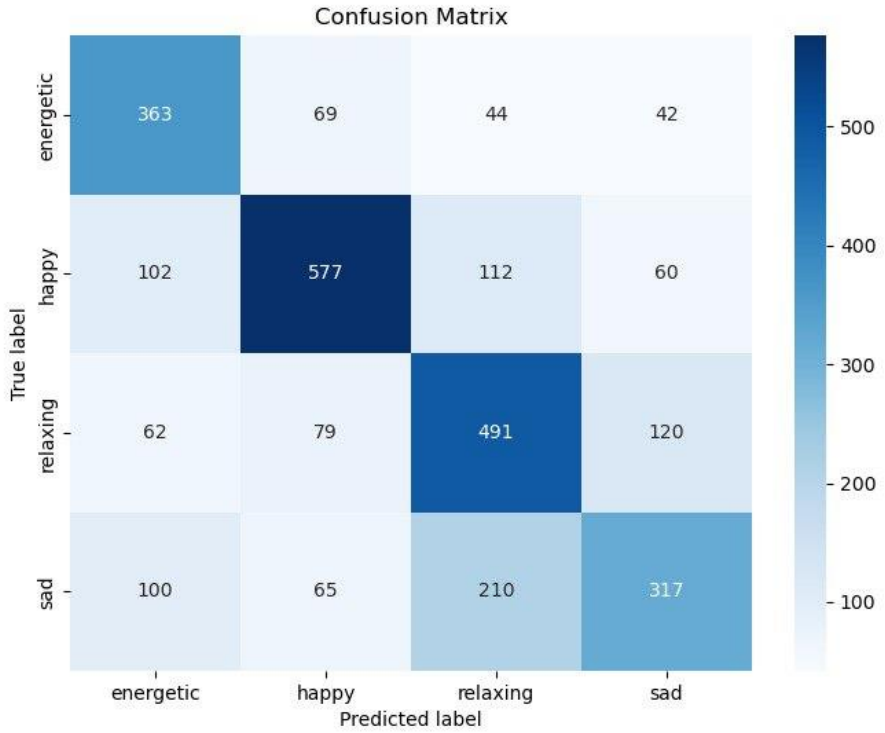



Рисунок 3.21 – Ошибки лучшей модели на 4-ех классах

Ввиду этого был изменен подход к задаче обучения: модели обучались на двух противоположных классах: «happy», «sad» и «relaxing», «energetic». Так был решена проблема перекрытия классов, и метрики качества модели поднялась с 0,6 до 0,7, как на тестовых, так и на тренировочных данных. Однако была выявлена следующая особенность: выбор между классами «relaxing» и «energetic» дается модели проще, чем выбор между классами «happy» и «sad». На рисунке 3.22 черным цветом изображен процесс обучения модели возможности отличать энергичную музыку от спокойной, а оранжевым цветом – грустную от веселой. Увеличение размерности срытого состояния модели лишь сгладило эту проблему.

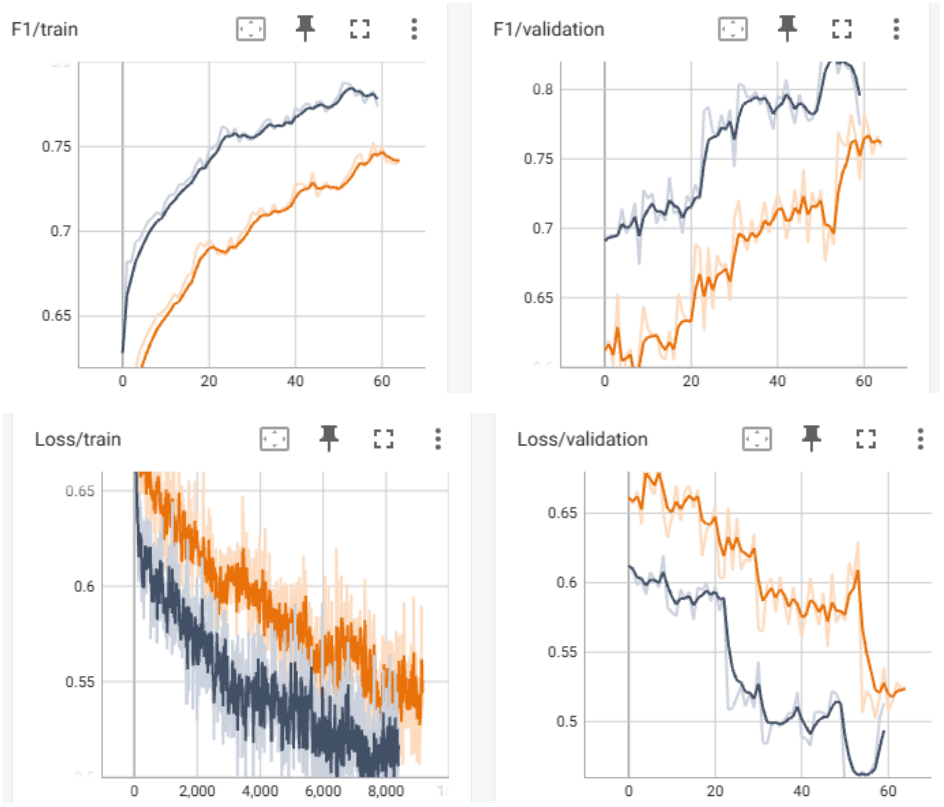



Рисунок 3.22 – Результаты обучения модели на классах «relaxing» и «energetic» лучше, чем результаты обучения на классах «happy» и «sad».

В результате тестирования модели на отложенной выборке были получены метрики: Precision ≈ 0,738; Recall ≈ 0,739; $F_1$ ≈ 0,738. Значение функции потерь составило 0,533 (усредненная оценка по мини-пакетам).

```
Total params: 7 929 522
Trainable params: 7 929 522

Evaluating model...
Test time: 47.495       loss: 0.533
Test precision: 0.738   recall: 0.739   F1: 0.738
```

Рисунок 3.23 – Результаты тестирования, классы «relaxing»/«energetic»

Матрица ошибок для этой модели представлена на рисунке 3.24.

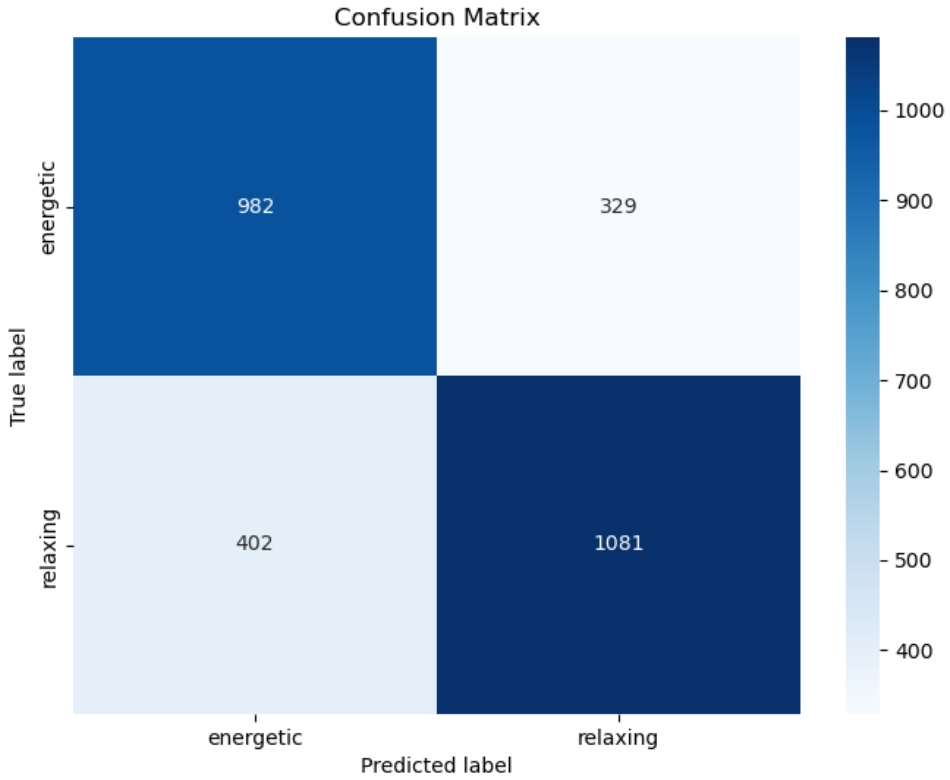



Рисунок 3.24 – Матрица ошибок на классах «relaxing» и «energetic»

Лучшая модель анализа спектрограмм для классификации «happy» / «sad» на тестовой выборке показала значения метрик: Precision ≈ 0,702; Recall ≈ 0,704; $F_1$ ≈ 0,698. Значение функции потерь составило 0,592.

```
Total params: 13 037 538
Trainable params: 13 037 538

Evaluating model...
Test time: 65.018       loss: 0.592
Test precision: 0.702   recall: 0.704   F1: 0.698
```

Рисунок 3.25 – Результаты тестирования, классы «happy»/«sad»

Матрица ошибок для этой модели представлена на рисунке 3.26.

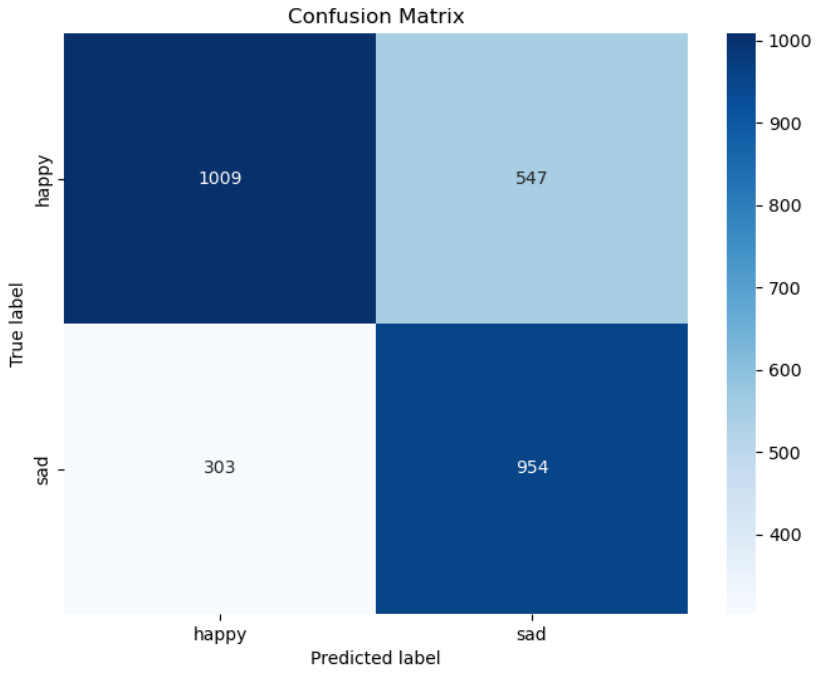



Рисунок 3.26 – Матрица ошибок на классах «happy» и «sad»

### 3.6 Обучение модели анализа звукового сигнала

С использованием обозначенных в предыдущем разделе подходов к улучшению качества обучения, были созданы модели для анализа звукового сигнала. В ходе экспериментов было установлено, что извлечение из необработанных аудиоданных характеристик, связанных с позитивностью музыки, является малоэффективным способом. В ходе обучения модели для решения такой задачи метрики качества не превышают отметки в 0,65. С другой стороны, определение энергичности музыки на основе звукового сигнала позволяет достичь значений метрик качества более 0,7. На рисунке 3.27 фиолетовым цветом обозначен график роста метрик для задачи классификации на метках «relaxing» и «energetic», а черным цветом – на метках «happy» и «sad».

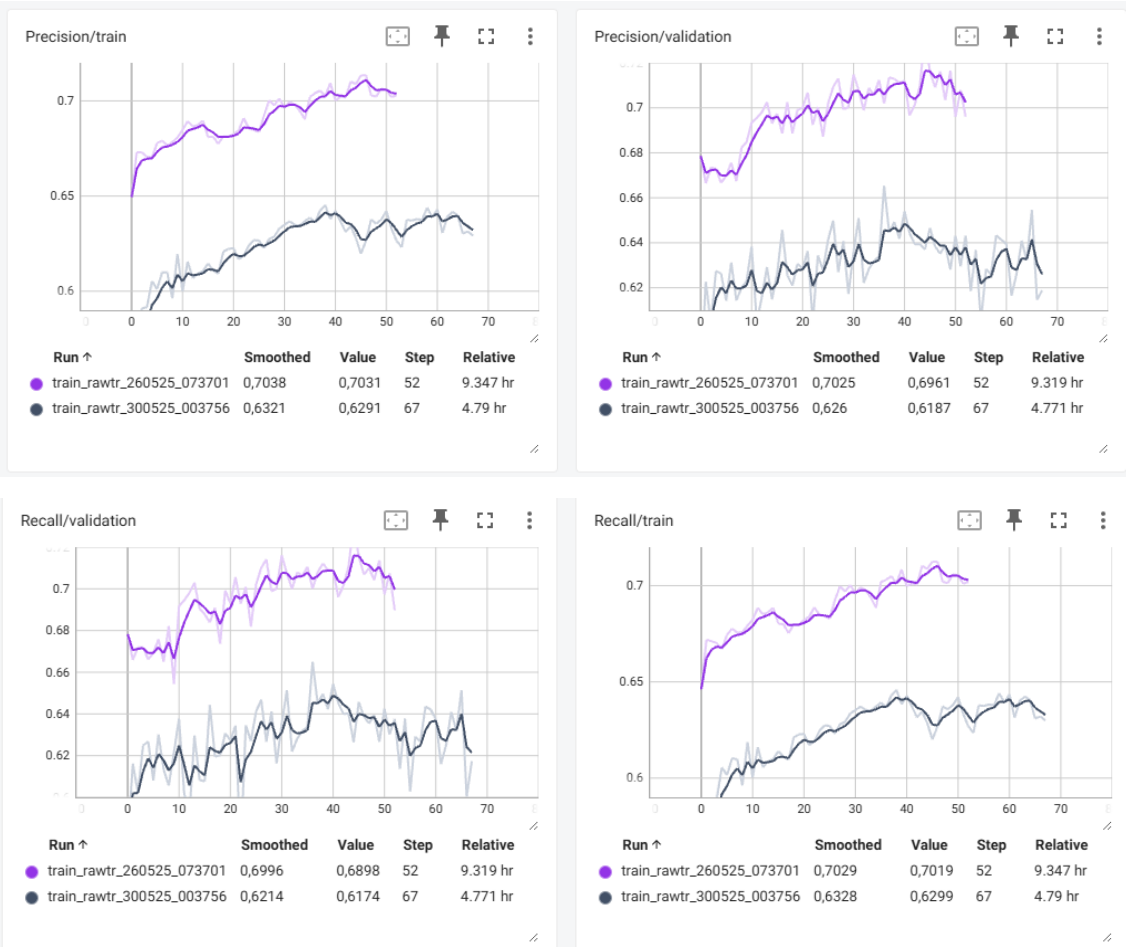



Рисунок 3.27 – Процесс обучения моделей анализа звукового сигнала

Лучшая модель анализа аудиоданных для классификации «relaxing» / «energetic» на тестовой выборке показала значения метрик: Precision ≈ 0,703; Recall ≈ 0,697; $F_1$ ≈ 0,697. Значение функции потерь составило 0,57.

```
Total params: 9 195 442
Trainable params: 9 195 442

Evaluating model...
Test time: 102.393       loss: 0.569
Test precision: 0.703    recall: 0.697   F1: 0.697
```

Рисунок 3.28 – Результаты тестирования модели анализа звукового сигнала, классы «relaxing»/«energetic»

Матрица ошибок для этой модели представлена на рисунке 3.29.

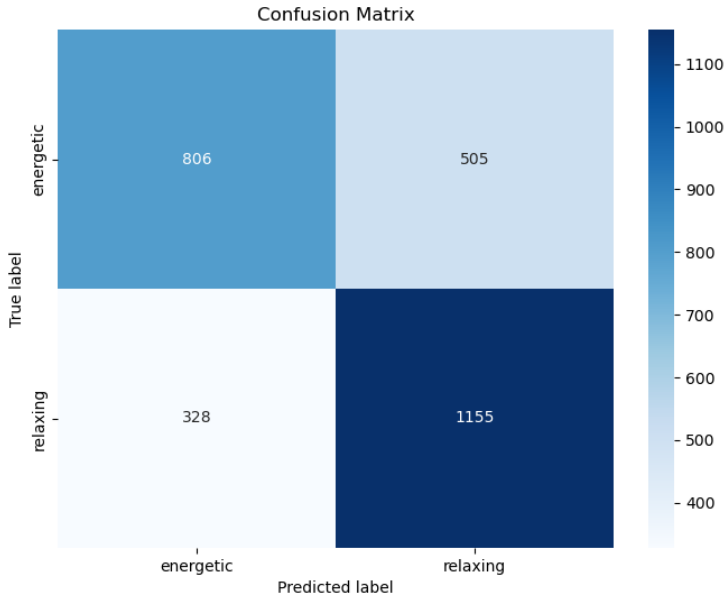



Рисунок 3.29 – Матрица ошибок модели анализа звукового сигнала, классы «relaxing»/«energetic»

Поскольку качество классификации «happy» / «sad» для подобных моделей оказалось не высоким, они не будут участвовать в построении более сложных моделей для этой задачи. Выход и веса лучшего классификатора для «relaxing»/«energetic», наоборот, может улучшить качество предсказаний.

### 3.7 Обучение модели анализа характеристик аудио

Для сравнения архитектур, представленных ранее, с конкурирующей им моделью для анализа векторных характеристик аудио, был создан и обучен MLP. Качество его предсказаний оказалось хуже, так как значения метрик для лучшей модели на тестовой выборке составили: Precision $\approx 0{,}644$; Recall $\approx 0{,}642$; $F_1 \approx 0{,}637$ (рисунок 3.30). Повышение сложности модели вело к неизбежному переобучению модели (рисунок 3.31). На матрице ошибок видно, что модель чаще отдает предпочтение классу «energetic» (рисунок 3.32).

```
Total params: 16 258
Trainable params: 16 258

Evaluating model...
Test time: 25.052        loss: 0.641
Test precision: 0.644    recall: 0.642   F1: 0.637
```

Рисунок 3.30 – Результаты тестирования MLP

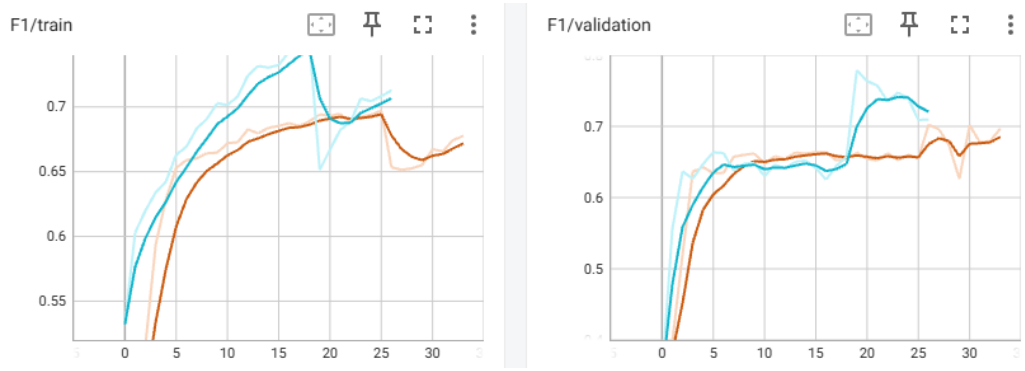



Рисунок 3.31 – Процесс обучения. Оранжевым цветом обозначен процесс обучения модели с 16 тыс. параметрами, синим – с 50-ю тысячами

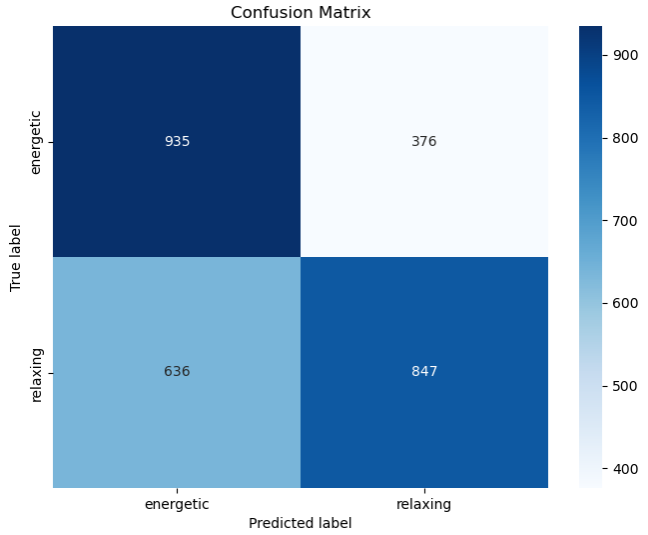



Рисунок 3.32 – Матрица ошибок для MLP

Кроме того, извлечение характеристик, необходимых для такой модели, является дорогостоящей по времени операцией. Ввиду этого, анализ векторных признаков звукового сигнала для построения более сложных моделей не будет использоваться вообще.

### 3.8 Сравнение лучших моделей с предварительно обученным Wav2Vec

Wav2Vec 2.0 является предварительно обученной моделью. Частично ее разработчики применяли обучение без учителя, маскируя около половины выхода кодирующей части трансформера и ставя задачу выбора правильного исходного представления данных на выходе декодирующей части [10]. Затем разработчики дополнительно обучали модель для задачи транскрибирования

речи. В результате кодирующую часть Wav2Vec можно использовать для построения эффективного представления необработанного звукового сигнала, с применением на его основе классификатора (лучшим выбором стал блок внимания с несколькими головами и MLP).

Однако Wav2Vec (как и многие другие крупные модели) обучался преимущественно на человеческой речи и в ходе экспериментов оказался малоэффективным в задачах классификации музыки. Значения функции потерь при обучении снижались очень медленно до 0,68 и не смогли достичь уровня качества моделей, обозреваемых ранее (рисунок 3.33).

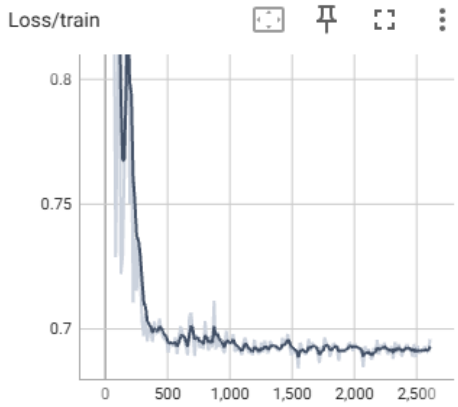



Рисунок 3.33 – Функция потерь в процессе обучения Wav2Vec с классификатором

Кроме того, дополнительное обучение такой модели требовало больших объемов видеопамяти и оказалось затратным по времени. На вычисление и применение градиентов для одного мини-пакета из 32 элементов Wav2Vec даже с выключенными из обучения весами требовал 2 секунды (примерно 9,4 минуты на эпоху). Модели на основе собственной архитектуры показывали производительность в 1 секунду на мини-пакет из 32 элементов (примерно 4,7 минут на эпоху).

### 3.9 Обучение модели анализа эмоциональной окраски текста

Построенная модель в ходе обучения показала быструю сходимость к решению, однако при помощи отложенной выборки был выявлен эффект

переобучения. Несмотря на принятые меры, описанные в разделе 3.5, обобщающая способность модели продолжала быть не высокой (рисунок 3.34). Для нивелирования этого эффекта архитектура была упрощена (внутренняя глубина трансформера снижена до 256, количество слоев и голов внимания до 4). В результате эффект переобучения удалось уменьшить (рисунок 3.35).

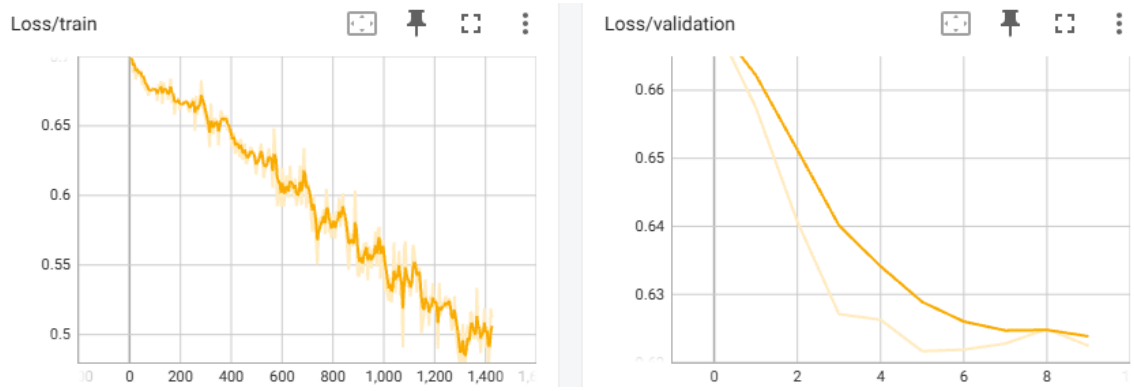



Рисунок 3.34 – Функция потерь в процессе обучения классификатора настроения текста

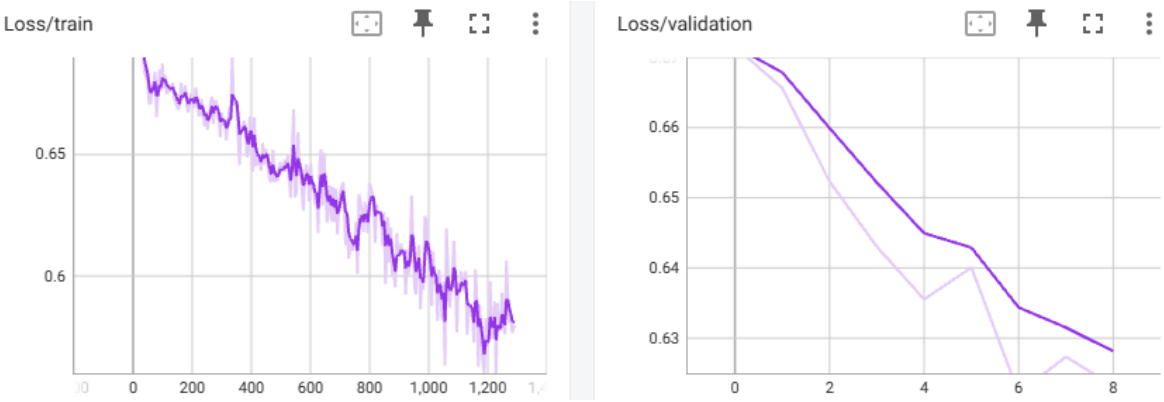



Рисунок 3.35 – Функция потерь после облегчения архитектуры

Далее полученная модель была объединена с предварительно обученной моделью Whisper. Поскольку основной набор данных в большем количестве содержит музыку без слов, для дополнительного обучения и проверки качества модели использовалась выборка из 100 песен с текстом. В ходе работы было выявлено, что извлечение токенов из аудио даже для малоразмерной версии Whisper является слишком затратным по времени:

обработка мини-пакета из 16 элементов занимает 10-15 секунд при условии «заморозки» весов Whisper и слоя Embedding.

Однако точность полученной модели оказалась высокой Precision ≈ 0,700; Recall ≈ 0,702; $F_1$ ≈ 0,690 (рисунок 3.36), но стоит учитывать, что обучение проходило на малой выборке.

```
Test time: 20.489       loss: 0.609
Test precision: 0.700   recall: 0.702   F1: 0.690
```

Рисунок 3.36 – Результаты тестирования модели для определения эмоциональной окраски текста

### 3.10 Построение, дополнительное обучение и тестирование моделей анализа гетерогенных данных

В ходе работы было создано несколько моделей машинного обучения, которые можно использовать для определения эмоциональной окраски музыкальных произведений. В таблице 2 приведен сравнительный анализ полученных моделей.

Таблица 2 – сравнительный анализ лучших моделей

| № п/п | Архитектура | Исходные данные | Целевые классы | Параметры | $F_1$ (test) |
|---|---|---|---|---|---|
| 1 | CNN → GRU → Transformer | Мел-спектрограммы | «relaxing» «energetic» | 8 млн. параметров, 5 слоев свертки, глубина GRU 256, трансформера – 432, 6 голов, 4 слоя | ~0,74 |
| 2 | CNN → GRU → Transformer | Мел-спектрограммы | «happy» «sad» | 13 млн. параметров, 5 слоев свертки, глубина GRU 256, трансформера – 432, 6 голов, 4 слоя | ~0,70 |
| 3 | CNN → GRU → Transformer | Звуковой сигнал | «relaxing» «energetic» | 9 млн. параметров, 7 слоев свертки, глубина GRU 256, трансформера – 312, 6 голов, 4 слоя | ~0,70 |
| 4 | Whisper → | Звуковой | «happy» | 276 млн. параметров (из | ~0,69 |

| | | | | | |
|---|---|---|---|---|---|
| | Transformer | сигнал → Текст | «sad» | них обучаемых – 21 млн.), глубина трансформера и Embedding слоя – 312, 6 голов, 4 слоя | |

Исходя из того, что увеличение количества разнородной информации, охватываемой моделью, может улучшить качество предсказаний, было принято решение объединить модели, обученные на одинаковых целевых классах.

Объединение происходит не на уровне выхода модели, а на уровне выхода блока внимания, как это описывалось в разделе 3.4. Веса моделей «замораживаются», обучается только классификатор на основании скрытых представлений.

Объединение моделей 1 и 3 не улучшило результат классификации. Значения метрик качества: $\mathrm{Precision} \approx 0{,}713$; $\mathrm{Recall} \approx 0{,}713$; $\mathrm{F_1} \approx 0{,}713$ (рисунок 3.37) лучше, чем значения тех же метрик для модели 3, но они не превосходят качество первой модели.

```
Total params: 17 325 442
Trainable params: 201 730

Evaluating model...
Test time: 94.124       loss: 0.564
Test precision: 0.713   recall: 0.713   F1: 0.713
```

Рисунок 3.37 – Результаты тестирования модели с двумя входящими тензорами: звуковым сигналом и мел-спектрограммой

Объединение моделей 2 и 4 позволило достичь самой высокой точности классификации $\mathrm{Precision} \approx 0{,}780$; $\mathrm{Recall} \approx 0{,}780$; $\mathrm{F_1} \approx 0{,}780$ (рисунок 3.38), однако стоит учитывать, что дополнительное обучение проводилось на маленьком, вручную размеченном наборе данных, поэтому шумов в данных при обучении было на порядок меньше.

```
Total params: 276 266 522
Trainable params: 5 312 746

Test time: 26.726        loss: 0.489
Test precision: 0.780    recall: 0.780   F1: 0.780
```

Рисунок 3.38 – Результаты тестирования модели с двумя входящими тензорами: текстом (из звукового сигнала) и мел-спектрограммой

Кроме того, извлечение текста из аудио слишком затратное по времени (пункт 3.9).

Таким образом, лучшей в задаче классификации музыки оказалась модель с архитектурой CNN → GRU → Transformer, обученная на спектрограммах. Она обеспечивает приемлемую точность и высокую скорость обработки данных (1 секунда на мини-пакет размером 32x96x5000). Поэтому для определения позитивности музыки следует использовать модель 2, а для определения энергичности музыки – модель 1.

### 3.11 Разработка и тестирование веб-приложения

Обученные модели были размещены и запущены в изолированном окружении, для них был создан API. Поскольку для выполнения предсказания используются обе полученные модели (для разных классов), в дальнейшем они не будут разделяться (будут называться одной моделью). При получении запроса на использование модели, алгоритм строит на основании аудиофайла пользователя мел-спектрограмму с параметрами, описанными в разделе 3.2.3. Если сам аудиофайл записан с частотой дискретизации большей, чем 16 кГц, то алгоритм выполняет повторную дискретизацию с приведением частоты к 16 кГц. Так гарантируется, что в процессе работы модель обрабатывает данные того же типа, на которых она обучалась. Затем модель последовательно получает на вход несколько окон из полученных спектрограмм и звуковых сигналов. Вероятностное распределение целевых классов для всех окон усредняется и возвращается клиенту.

Серверное приложение было размещено во второй изолированной среде выполнения, была настроена маршрутизация между ним, клиентской частью и API модели. Была реализована логика обработчиков запросов: для предсказания с использованием ссылки и с использованием пользовательского файла в форматах MP3 или WAV. Были установлены ограничение на максимальный размер загружаемого файла (50 МБ) и на частоту запросов (не чаще одного в минуту).

С помощью библиотеки React был создан пользовательский интерфейс, состоящий из одной страницы и двух всплывающих окон. Внешний вид домашней страницы представлен на рисунке 3.39.

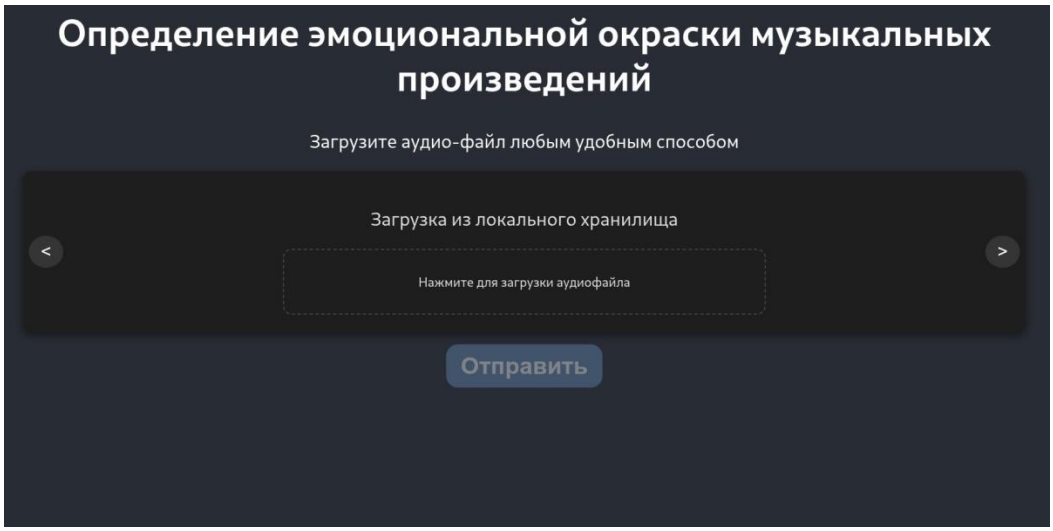



Рисунок 3.39 – Домашняя страница web-приложения

Для пользователя предусмотрена возможность загрузки аудиофайла из локального хранилища (рисунки 3.40-3.41) или по ссылке с онлайн ресурса Jamendo (рисунки 3.42-3.43). Предусмотрены проверки вводимых пользователем данных как на стороне клиента (рисунок 3.44), так и на стороне сервера.

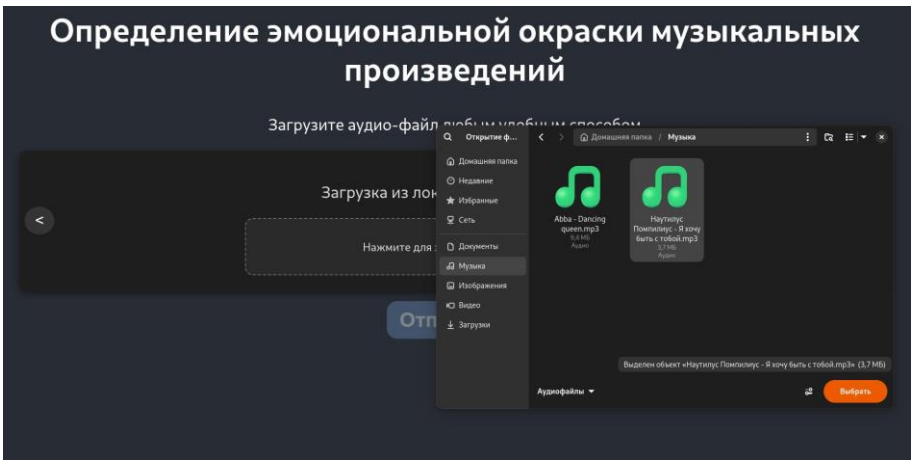



Рисунок 3.40 – Загрузка аудиофайла из локального хранилища

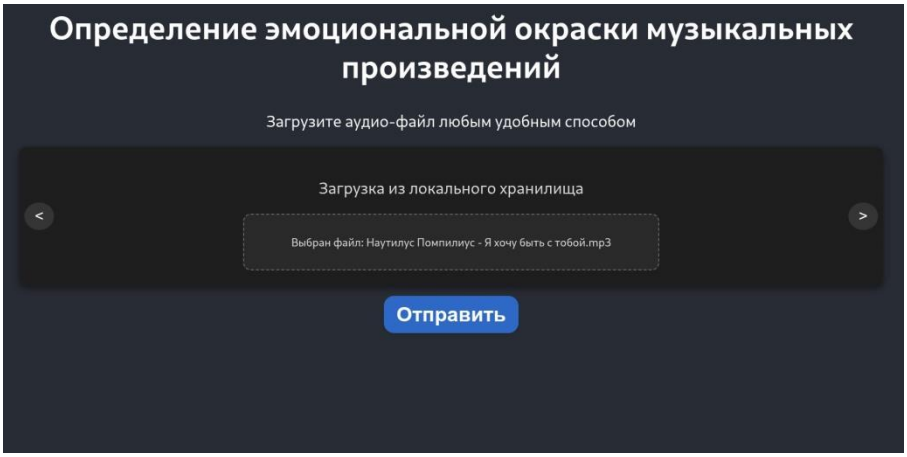



Рисунок 3.41 – Результат загрузки файла

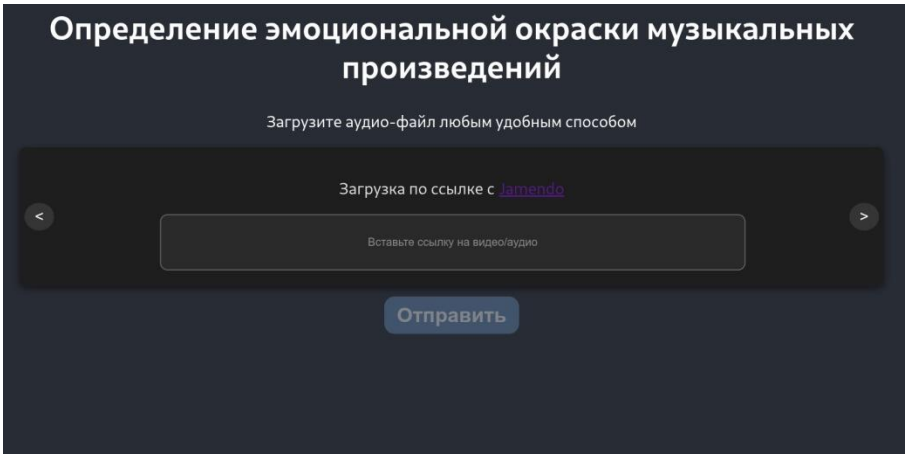



Рисунок 3.42 – Ввод ссылки на композицию

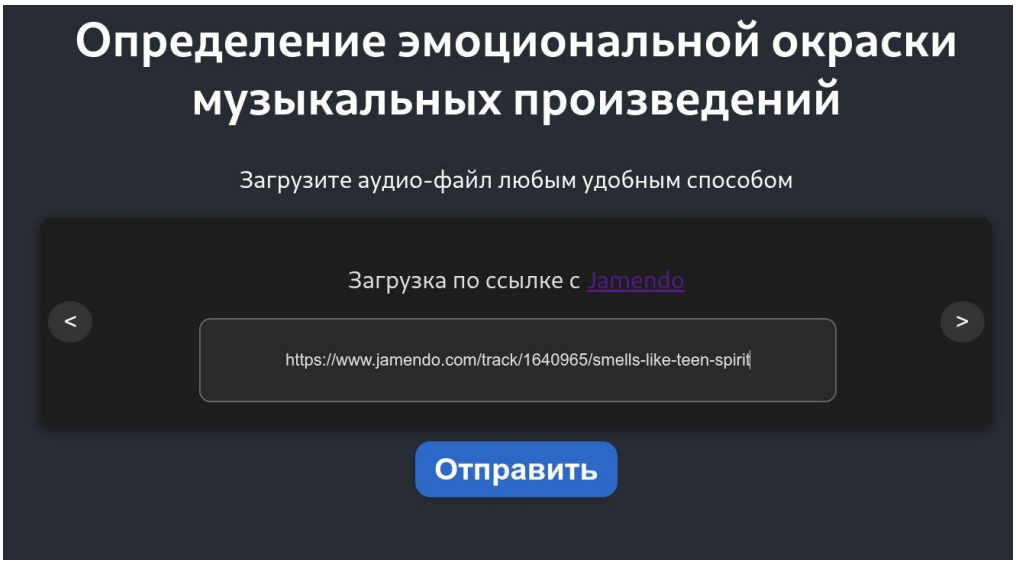



Рисунок 3.43 – Корректный ввод ссылки

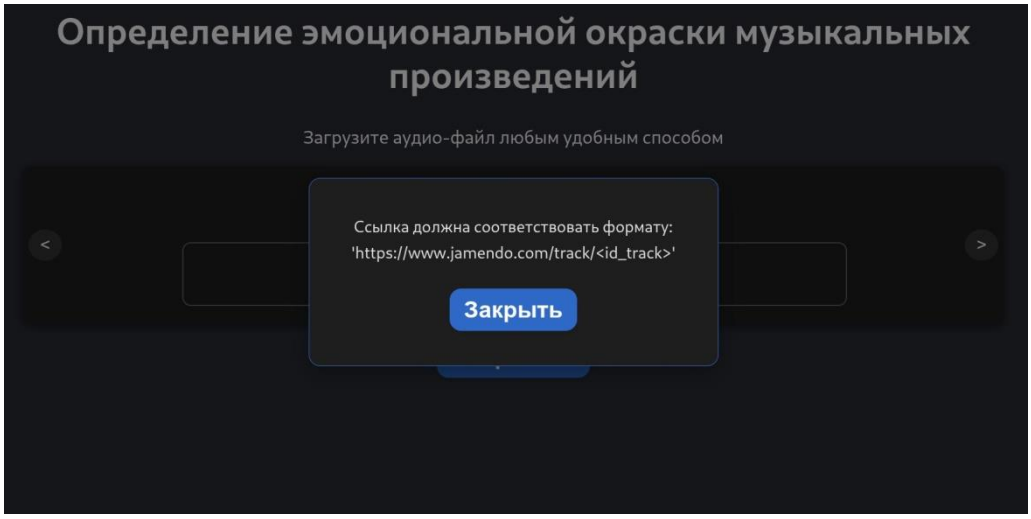



Рисунок 3.44 – Предупреждение об ошибке после некорректного ввода ссылки

Результат классификации отображается во всплывающем окне (рисунки 3.45 и 3.46), пользователь может оценить вероятностное распределение классов при помощи сетчатой диаграммы. Вверху окна отображается доминирующий класс. При этом просмотр результата сопровождается цветовым изменением некоторых элементов интерфейса.

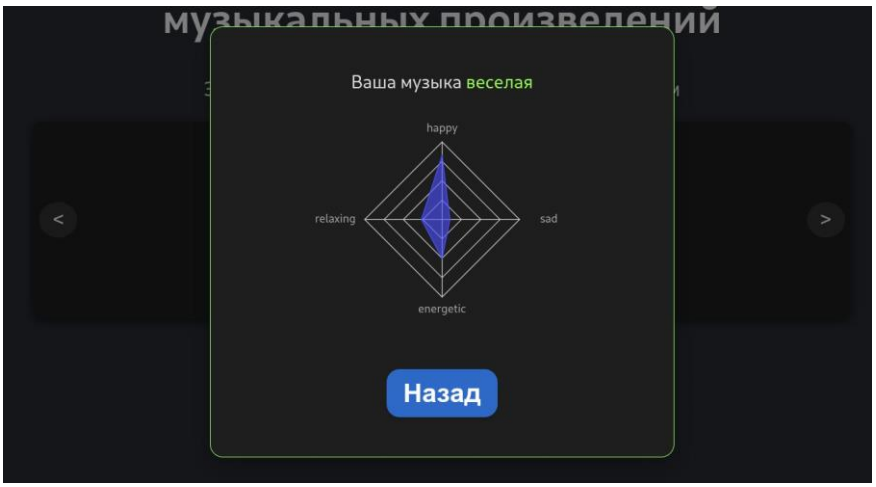



Рисунок 3.45 – Результат оценки модели (композиция «Dancing Queen», ABBA)

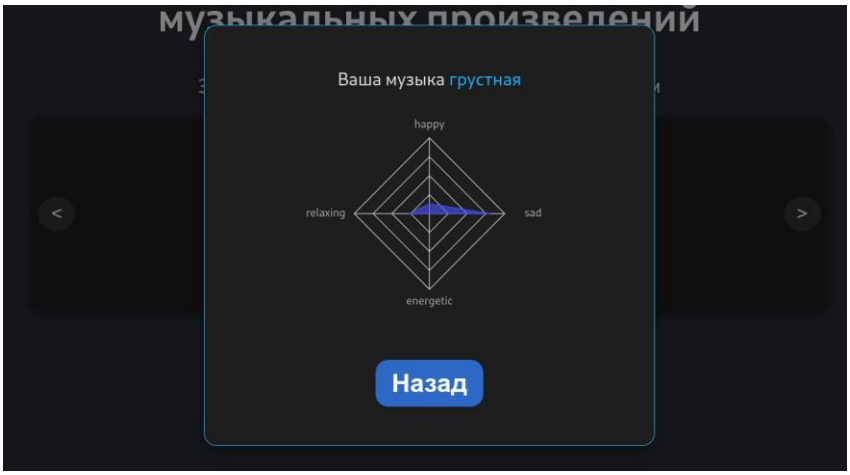



Рисунок 3.46 – Результат оценки модели (композиция «Я хочу быть с тобой», Наутилус Помпилиус)

Поскольку использовалась адаптивная верстка, приложение можно использовать на мобильных устройствах (рисунок 3.47).

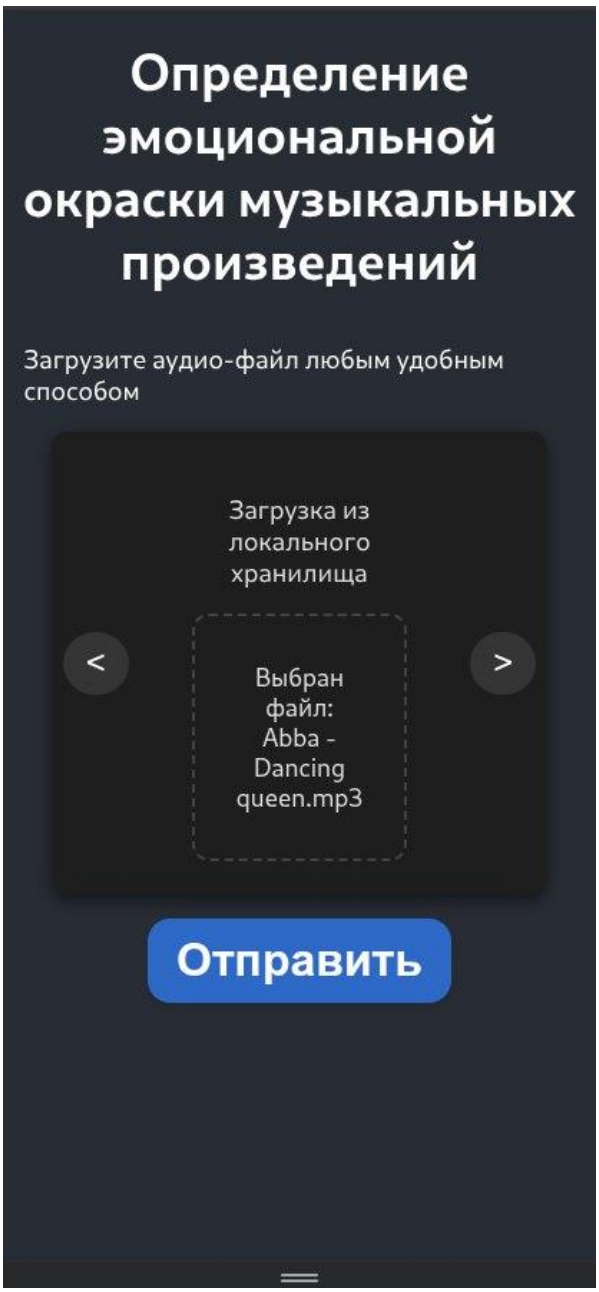



Рисунок 3.47 – Адаптивная верстка

# ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы были проведены анализ предметной области и исследование современных технологий, инструментов и моделей. С использованием полученных знаний были выбраны лучшие подходы к обработке данных, построению и обучению моделей, созданию клиент-серверных приложений.

Были получены, обработаны и исследованы различные входные данные для моделей машинного обучения: звуковой сигнал, мел-спектрограммы, текст песен и характеристики аудио. На основании этого анализа было установлено, что наилучшие результаты классификации достигаются при использовании мел-спектрограмм в качестве входных признаков (раздел 3.10).

В результате была создана модель глубоких искусственных нейронных сетей на основе конвейера CNN → GRU → Transformer, обеспечивающая качество классификации, превосходящее более простые модели, например MLP. Значения метрик лучшей модели MLP на задаче определения энергичности музыки составляют $\text{Precision} \approx \text{Recall} \approx \text{F}_1 \approx 0{,}64$, в то время как точность разработанной модели на 15% выше: $\text{Precision} \approx \text{Recall} \approx \text{F}_1 \approx 0{,}74$ (разделы 3.5 и 3.7).

При этом временные затраты на обучение и предсказание ниже, чем у предварительно обученных моделей в 2 раза (раздел 3.8). Таким образом, готовый продукт соответствует заявленным требованиям к быстродействию. Кроме того, точность собственной разработки выше, чем точность предварительно обученных моделей – значения функции потерь составили 0,53 и 0,68 для разработанной модели и для предварительно обученного Wav2Vec 2 соответственно.

Исследование также показало, что использование разнородных данных как единого входного тензора для модели может повышать ее точность, но

часто ведет к ухудшению производительности. В этом направлении перспективным является дальнейшее развитие архитектур, способных интегрировать различные источники информации при сохранении вычислительной легкости.

Разработанная модель была внедрена в программную систему для определения эмоциональной окраски музыки на основе аудиофайла. Система обладает отказоустойчивой архитектурой, пользовательским интерфейсом, потенциалом для расширения.

Полученные результаты и решения позволяют использовать систему, как в практических целях – для автоматической разметки больших музыкальных коллекций, так и в исследовательских целях – для последующего улучшения моделей за счет дополнительного обучения на других наборах данных и задачах.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Исследование GfK за 3-й квартал 2024 // ICT Online URL: https://releases.ict-online.ru/news/Issledovaniye-GfK-za-3-i-kvartal-2024-pochti-kazhdyi-vtoroi-gorozhanin-pol-zuyet-sya-podpiskami-na-muzykal-nyye-strimingi-301198 (дата обращения: 20.05.2025)

2. Пресс-релиз: Главные итоги VK за первый квартал 2025 года // МКПАО «ВК» URL: https://corp.vkcdn.ru/media/files/RUS_Press_Release_Q1_2025.pdf (дата обращения: 20.05.2025)

3. Яндекс объявляет финансовые результаты за I квартал 2025 года // МКПАО «Яндекс» URL: https://yastatic.net/s3/ir-docs/docs/2025/q1/38e8f62230gc3c915f300u6y4312c63f/[RUS]%20MKPAO_Q1_2025_30gc.pdf (дата обращения: 20.05.2025)

4. **Николенко С**. Глубокое обучение / Николенко С., Кадурин А., Архангельская Е. - СПб.: Питер, 2019. - 480 с.

5. **Гудфеллоу И.** Глубокое обучение: пер. с англ. / И. Гудфеллоу, Й. Бенджио, А. Курвил; науч. Ред. Пер. В. В. Стрижов. – М.: Диалектика, 2018. – 656 с.

6. Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L., Polosukhin I. Attention Is All You Need // arXiv – URL: https://arxiv.org/pdf/1706.03762 - Дата публикации: 12.06.2017;

7. Трансформеры // Яндекс Образование URL: https://education.yandex.ru/handbook/ml/article/transformery (дата обращения 05.04.2025)

8. Набор данных MTG-Jamendo // GitHub MTG/mtg-jamendo-dataset URL: https://github.com/MTG/mtg-jamendo-dataset (дата обращения: 15.02.2025)

9. **А. Б. Сергиенко**. Цифровая обработка сигналов: учеб. пособие. – 3-е изд. – СПб.: БВХ-Петербург, 2011. – 768 с.

10. Baevski A., Zhou H., Mohamed A., Auli M. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations // arXiv – URL: https://arxiv.org/pdf/2006.11477 - Дата публикации: 22.10.2020

11. Преобразование Фурье: самый подробный разбор // Библиотека программиста URL: https://proglib.io/p/fourier-transform (дата обращения: 20.03.2025)

12. Radford A., Kim J. W., Xu T., Brockman G., McLeavey C., Sutskever I. Robust Speech Recognition via Large-Scale Weak Supervision // arXiv – URL: https://arxiv.org/pdf/2212.04356 - Дата публикации: 06.12.2022

13. Набор данных XED // GitHub URL: https://github.com/Helsinki-NLP/XED/tree/master (дата обращения: 16.03.2025)

14. Gulati A., Qin J., Chiu C., Parmar N., Zhang Y., Yu J., Han W., Wang S., Zhang Z., Wu Y., Pang R. Conformer: Convolution-augmented Transformer for Speech Recognition // arXiv – URL: https://arxiv.org/pdf/2005.08100 - Дата публикации: 16.06.2020

15. Архитектуры трансформеров для аудио // Hugging Face URL: https://huggingface.co/learn/audio-course/ru/chapter3/introduction (дата обращения: 18.03.2025)

16. Jhanji E. EXPLORING AND APPLYING AUDIO-BASED SENTIMENT ANALYSIS IN MUSIC // arXiv – URL: https://arxiv.org/pdf/2403.17379 - Дата публикации: 22.02.2024

17. Niizumi D., Takeuchi D., Ohishi Y., Harada N., Kashino K. Masked Spectrogram Modeling using Masked Autoencoders for Learning General-purpose Audio Representation // arXiv – URL: https://arxiv.org/pdf/2204.12260 Дата публикации: 26.04.2022

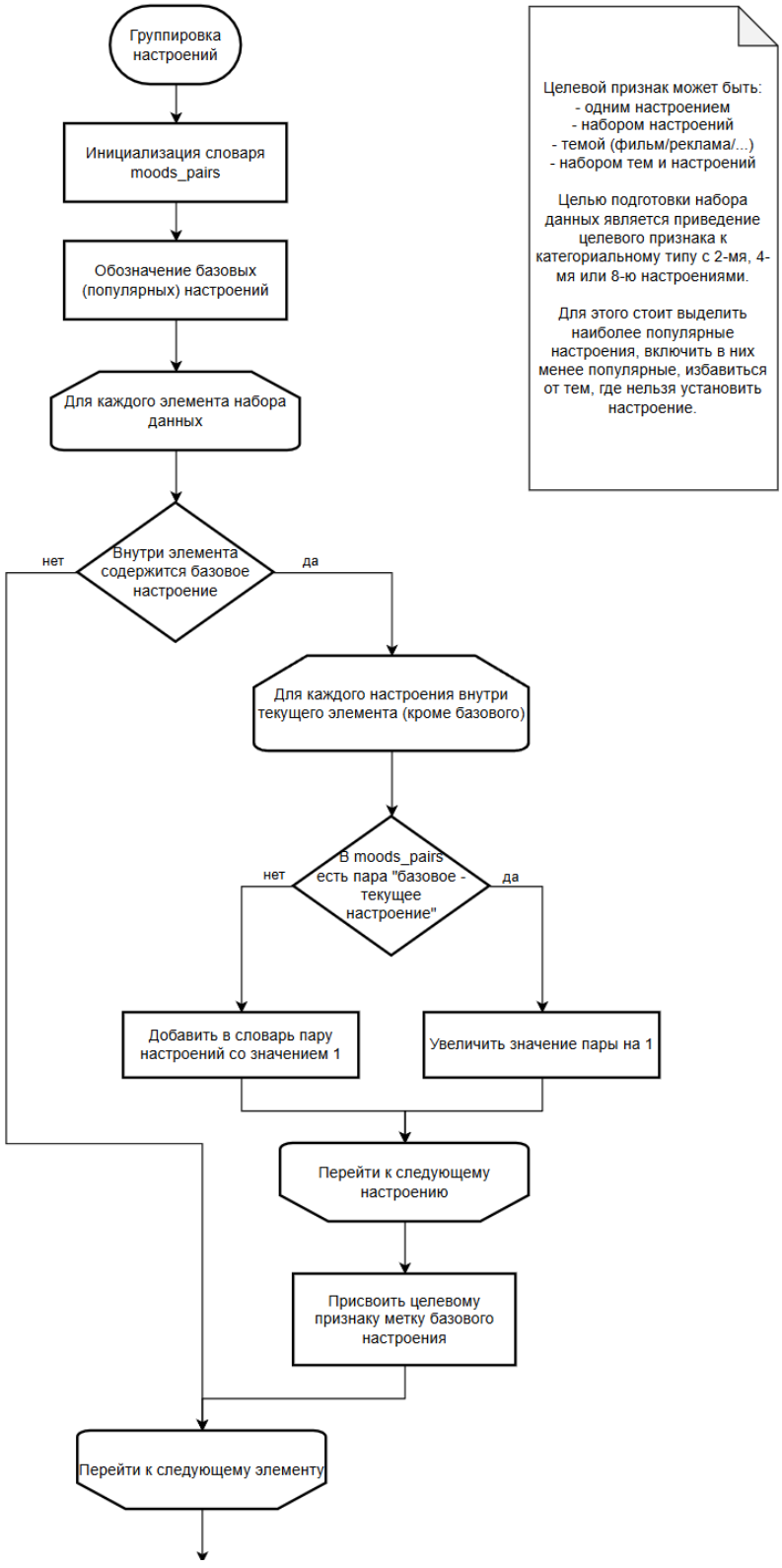
18. Документация Python 3.10 // Python docs URL: https://docs.python.org/release/3.10.0/ (дата обращения: 27.03.2025)

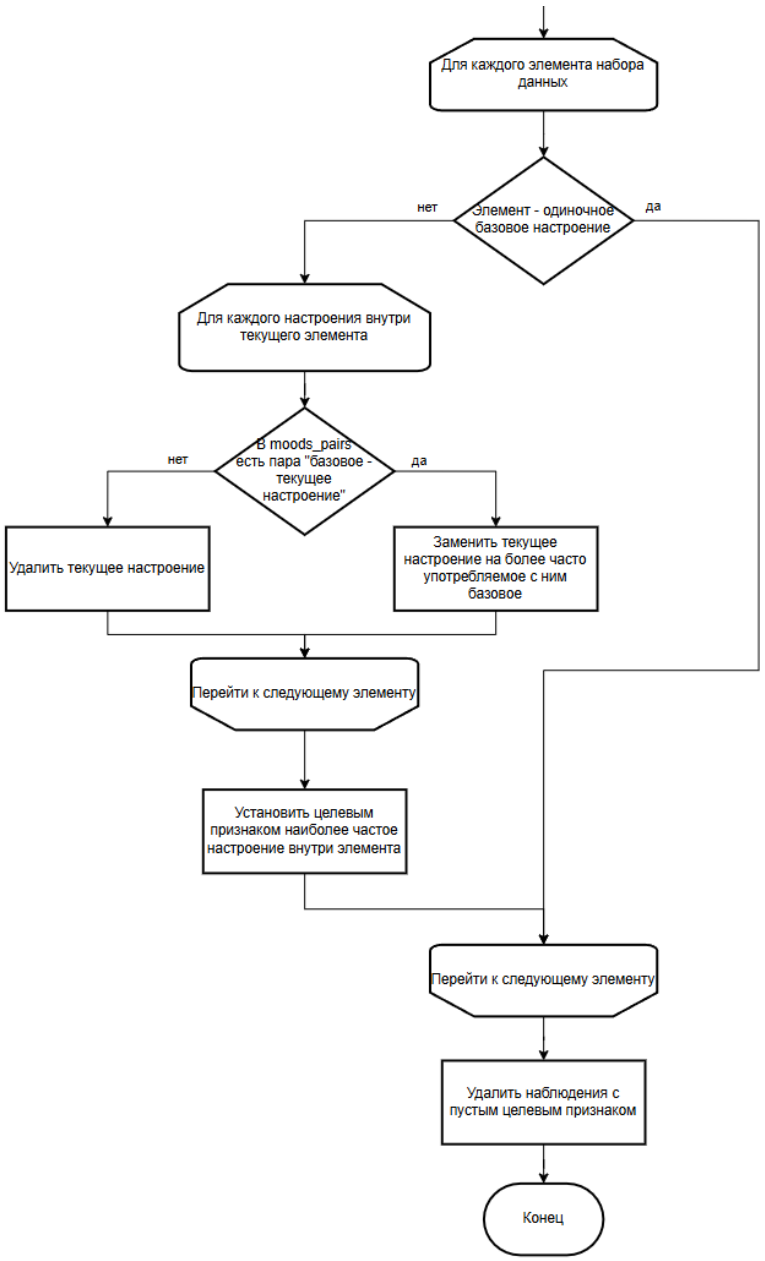
19. Документация PyTorch для разработчиков // PyTorch URL: https://docs.pytorch.org/docs/stable/index.html (дата обращения: 01.04.2025)

20. 7 методов для анализа настроения аудио // AIMultiple Research URL: https://research.aimultiple.com/audio-sentiment-analysis/ (дата обращения: 05.04.2025)

21. Документация React для разработчиков // React dev URL: https://reactdev.ru/ (дата обращения: 14.05.2025)

# ПРИЛОЖЕНИЕ А – Блок-схема алгоритма предварительной обработки данных и слияния целевых меток






На этом этапе целевой признак может быть:
- базовым настроением
- не базовым настроением или темой
- набором не базовых настроений и тем

Поскольку был составлен словарь с частотами пар "базовое настроение - не базовое настроение/тема", то на основании него можно заменить не базовые настроения и темы на базовые, полагаясь на то, что чем чаще настроение было записано с другим, тем больше у них общего.

В результате в наборе могут остаться неразмеченные данные. Такие наблюдения следует либо удалить, либо разметить вручную

# ПРИЛОЖЕНИЕ Б – Листинг программы предварительной обработки данных

```
import os
import ast
import json
import pandas as pd
from dotenv import load_dotenv
from collections import Counter
from argparse import ArgumentParser
from exceptions.exceptions import InvalidConfigException


def cli_arguments_preprocess() -> tuple:
    """
    Read, parse and preprocess command line arguments:
        - path to source dataset. Required
        - path to mel spectrograms. Should be related to dataset path. By default:
"melspecs/"
        - number of moods. By default: all moods (0)
        - use features dataset as source dataset

    Sorce dataset file should be named "autotagging_moodtheme.tsv"
    """
    parser = ArgumentParser(description="Preprocessing autotagging moods dataset
script. Cleans moods, aggregates them and saves preprocessed .tsv file")

    parser.add_argument("--path", required=True,
                        help="Path to source dataset")

    parser.add_argument("--name", required=True,
                        help="Name of the source dataset")

    parser.add_argument("--mels", required=False,
                        help="Path to mel spectrograms. Should be related to the da-
taset path. By default: 'melspecs/'")

    parser.add_argument("--moods", required=False,
                        choices=["2", "hs", "re", "4", "8", "all"],
                        help="Number of aggregated moods or names of agregated moods.
By default: all source moods")

    parser.add_argument("-f", "--features", action="store_true", help="Use features
daaset as the source dataset")

    args = parser.parse_args()

    if args.path[-1] != "/":
        args.path += "/"

    if args.mels is None:
        args.mels = "melspecs/"

    if not args.moods:
        args.moods = "all"
```

```
    return os.path.abspath(args.path), args.name, args.mels, args.moods,
args.features

def load_dataset(dataset_path: str) -> pd.DataFrame:
    """
    Reads the dataset from the specified path and returns it as a pandas DataFrame.
    Handles cases where the last column contains multiple values separated by tabs.
    """
    try:
        # Read the dataset without splitting the last column.
        with open(dataset_path, "r", encoding="utf-8") as file:
            lines = file.readlines()

        columns_number = len(lines[0].split("\t"))

        # Split lines by tabs; last column may contain multiple values.
        data = [line.strip().split("\t", maxsplit=columns_number - 1) for line in
lines]

        # Create a DataFrame
        df = pd.DataFrame(data[1:], columns=data[0])  # First row as header
        return df
    except FileNotFoundError:
        print(f"File not found: {dataset_path}")
        return None
    except Exception as e:
        print(f"An error occurred: {e}")
        return None

def load_features_dataset(dataset_path: str) -> pd.DataFrame:
    """
    Reads the dataset from the specified path and returns it as a pandas DataFrame.
    """
    return pd.read_csv(dataset_path, sep="\t")

def clean_target(df: pd.DataFrame) -> pd.DataFrame:
    """
    Cleans the last column of the DataFrame by removing the prefix 'mood/theme---'
    and splites multi tags by tabs
    """
    last_column = df.columns[-1]
    df[last_column] = df[last_column].str.replace("mood/theme---", "", regex=False)
    df[last_column] = df[last_column].str.split("\t")
    return df

def clean_columns_name(df: pd.DataFrame) -> pd.DataFrame:
    """
    Refactors the column names to lower case
    """
    columns = df.columns
    refactored_columns = columns.str.lower()
    df.columns = refactored_columns
    return df

def clean_id_columns(df: pd.DataFrame) -> pd.DataFrame:
    """
    Cleans the ID columns by removing the prifix "track", "artist", "album" and pars-
es them to int
```

```
    Remember that this code cleanes first three columns: "track_id", "artist_id",
"album_id" in such order
    """
    track_id_column = df.columns[0]
    artist_id_column = df.columns[1]
    album_id_column = df.columns[2]

    df[track_id_column] = df[track_id_column].str.replace("track_", "", regex=False)
    df[artist_id_column] = df[artist_id_column].str.replace("artist_", "", re-
gex=False)
    df[album_id_column] = df[album_id_column].str.replace("album_", "", regex=False)

    df[track_id_column] = pd.to_numeric(df[track_id_column])
    df[artist_id_column] = pd.to_numeric(df[artist_id_column])
    df[album_id_column] = pd.to_numeric(df[album_id_column])

    return df

def add_melspecs_path(df: pd.DataFrame, melspecs_rel_path: str) -> pd.DataFrame:
    """
    Adds the mel spectrograms path to the DataFrame.
    """
    # Add the mel spectrograms path to the DataFrame and reorder columns
    df.insert(df.columns.get_loc("path") + 1, "melspecs_path",
              df["path"].apply(lambda x: os.path.join(melspecs_rel_path,
x.replace(".mp3", ".npy"))))
    return df

def clean_dataset_pipeline(dataset: pd.DataFrame) -> pd.DataFrame:
    """
    Cleans the dataset by applying all cleaning functions in order.
    """
    df = dataset.copy()  # Avoid modifying the original DataFrame
    df = clean_columns_name(df)
    df = clean_target(df)
    df = clean_id_columns(df)
    return df

def merge_moods(dataset: pd.DataFrame, basic_moods: list, ban_moods: list) -> tuple:
    """
    Merges moods tags in several basic moods.

    Each unpopular mood merges with the basic mood that most often stands in pairs
with it.
    Removes ban moods from the tags. If the tag can't be merged with any basic mood,
row will be removed from the dataset.
    """
    mood_pairs = {}  # Set of basic moods sets of them pairs with pairs counts.
    non_basic_moods = set(dataset["tags"].explode().unique())  # Set of non basic
moods.

    for basic_mood in basic_moods:
        if basic_mood not in non_basic_moods:
            raise InvalidConfigException(f"Basic mood {basic_mood} from config not
found in dataset. Please check the config file.")

        if basic_mood in ban_moods:
```

```
            raise InvalidConfigException(f"Basic mood {basic_mood} found in ban list
{ban_moods}. Please check the config file.")

        mood_pairs[basic_mood] = {}
        non_basic_moods.remove(basic_mood)

    # Counting cycle to determine mood pairs.
    for i, row in dataset.iterrows():
        tags = row["tags"]
        current_basic_mood = None

        # Try to find basic mood in tags.
        for basic_mood in basic_moods:
            if basic_mood in tags:
                current_basic_mood = basic_mood
                break

        # If no basic mood in tags, skip this row.
        if current_basic_mood is None:
            continue

        # There is basic mood in tags, so we can build mood pairs.
        for tag in tags:
            if tag != current_basic_mood:
                # If tag is not in basic moods set, we can add it to the set with
count 1. Else -> increase count by 1.
                if tag in mood_pairs[current_basic_mood]:
                    mood_pairs[current_basic_mood][tag] += 1
                else:
                    mood_pairs[current_basic_mood][tag] = 1

        # Finally, replace current tag with basic mood.
        dataset.loc[i, "tags"] = [current_basic_mood]

    # Count the most popular neighbour mood for each non basic mood.
    moods_conformity = {}

    for mood in non_basic_moods:
        # Skip banned moods.
        if mood in ban_moods:
            moods_conformity[mood] = None
            continue

        pairs_counts = {}

        for basic_mood in mood_pairs.keys():
            # Remember pairs counts if tag was paired with basic mood.
            if mood in mood_pairs[basic_mood]:
                pairs_counts[basic_mood] = mood_pairs[basic_mood][mood]

        # Choose the most popular neighbour basic mood from the pairs counts.
        selected_mood = max(pairs_counts, key=pairs_counts.get) if pairs_counts else
None

        # Remember conformity.
        if selected_mood is not None:
            moods_conformity[mood] = selected_mood
        else:
```

```
            moods_conformity[mood] = None

    # Merging cycle to merge moods.
    for i, row in dataset.iterrows():
        tags = row["tags"]

        # In this case we have already merged moods in the dataset.
        if tags[0] in basic_moods:
            continue

        # Now we can start merge other moods.
        tags_to_remove = []
        for tag in tags:
            if moods_conformity[tag] is not None:
                tags[tags.index(tag)] = moods_conformity[tag]
            else:
                tags_to_remove.append(tag)

        for tag in tags_to_remove:
            tags.remove(tag)

        # Now select the most popular mood in the tags list.
        if len(tags) == 0 or len(tags) == 1:
            continue

        counter = Counter(tags)
        selected_mood = max(counter, key=counter.get)
        dataset.loc[i, "tags"] = [selected_mood]

    # Remove rows with empty tags and transform the tags-array with only one element
into a string
    # (it is guaranteed that tags contain only single-element arrays).
    dataset = dataset[dataset["tags"].apply(lambda x: len(x) > 0)].assign(tags=lambda
df: df["tags"].str[0])

    return dataset, mood_pairs, moods_conformity

def main():
    # Load dataset, environment and read cli arguments.
    dataset_path, dataset_source_name, melspecs_rel_path, moods_merge_mode,
use_features_dataset = cli_arguments_preprocess()
    load_dotenv()

    # Get required info from environment variables.
    config_path = os.getenv("CONFIG_PATH")
    outputs_path = os.getenv("OUTPUTS_PATH")

    if not os.path.exists(outputs_path):
        os.mkdir(outputs_path)

    # Load dataset.
    if use_features_dataset:
        dataset = load_features_dataset(os.path.join(dataset_path, da-
taset_source_name))
    else:
        dataset = load_dataset(os.path.join(dataset_path, dataset_source_name))

    if dataset is not None and not dataset.empty:
```

```
        # Clean dataset.
        print(f"Dataset loaded successfully with {len(dataset)} rows and
{len(dataset.columns)} columns.")
        print(dataset.head(n=3), "\n")

        # Features dataset already cleaned
        if not use_features_dataset:
            cleaned_dataset = clean_dataset_pipeline(dataset)

            print("Dataset cleaned successfully:")
            print(cleaned_dataset.head(n=3), "\n")
            cleaned_dataset.info()

            print("Add mel spectrograms path to the dataset.")
            cleaned_dataset = add_melspecs_path(cleaned_dataset, melspecs_rel_path)
            print(cleaned_dataset.head(n=3), "\n")

            # Get target tags distribution (for next merging).
            tags_distribution_save_path = os.path.join(outputs_path,
"tags_distribution.csv")
            tags_distribution = cleaned_dataset["tags"].explode().value_counts()
            tags_distribution.to_csv(tags_distribution_save_path, index=True)
            print(f"Tags distribution saved successfully. Path:
{tags_distribution_save_path}\n")

            save_path = os.path.join(dataset_path,
f"dataset_{moods_merge_mode}_moods.tsv")
        else:
            cleaned_dataset = dataset
            cleaned_dataset["tags"] = cleaned_dataset["tags"].apply(ast.literal_eval)
            save_path = os.path.join(dataset_path,
f"features_dataset_{moods_merge_mode}_moods.tsv")

        # If user defined moods mode as "all" then script should just save the
cleaned data.
        if moods_merge_mode == "all":
            cleaned_dataset.to_csv(save_path, sep="\t", index=False)
            print("Cleaned dataset saved successfully. Path: ", save_path)
            return

        # Load config to merge moods.
        with open(os.path.join(config_path, "moods.json")) as file:
            config = json.load(file)
            base_moods = config[f"mode_{moods_merge_mode}"]
            ban_moods = config[f"ban_list_{moods_merge_mode}"]

        # Merge moods in the dataset.
        final_dataset, moods_pairs, moods_conformity = merge_moods(cleaned_dataset,
base_moods, ban_moods)

        # Save moods pairs to json file.
        moods_pairs_save_path = os.path.join(outputs_path,
f"moods_pairs_{moods_merge_mode}.json")
        moods_conformity_save_path = os.path.join(outputs_path,
f"moods_conformity_{moods_merge_mode}.json")

        with open(moods_pairs_save_path, "w", encoding="utf-8") as file:
            json.dump(moods_pairs, file, indent=4)
```

```
        print("Moods pairs saved successfully to the output directory.")

        with open(moods_conformity_save_path, "w", encoding="utf-8") as file:
            json.dump(moods_conformity, file, indent=4)
        print("Moods conformity saved successfully to the output directory.\n")

        print(f"Final dataset has {len(final_dataset)} rows and
{len(final_dataset.columns)} columns.\n")
        print(final_dataset.head(n=10), "\n")
        final_dataset.info()

        tags_distribution = final_dataset["tags"].value_counts()
        print(f"\nTags distribution after merging moods:\n{tags_distribution}\n")

        # Save final dataset.
        final_dataset.to_csv(save_path, sep="\t", index=False)
        print("Dataset saved successfully. Path: ", save_path)
    else:
        print("Failed to load dataset.")


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

# ПРИЛОЖЕНИЕ В – Листинг обучающего цикла

```
import os
import re
import torch
from time import time
from datetime import datetime
import torch.optim as optim
from torch.utils.tensorboard.writer import SummaryWriter
from torchmetrics.classification import (
    MulticlassPrecision, MulticlassRecall, MulticlassF1Score,
    MultilabelPrecision, MultilabelRecall, MultilabelF1Score
)

import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

from config import *
from model.data import KFoldSpecsDataLoader


class EarlyStopping:
    def __init__(self, patience: int = 5, min_delta: float = 0.0):
        """
        :param patience: epoch count, after which training will be stopped if no im-
provement
        :param min_delta: minimal loss delta to consider improvement
        """
        self.patience = patience
        self.min_delta = min_delta
        self.best_loss = float('inf')
        self.counter = 0
        self.should_stop = False

    def step(self, current_loss: float):
        if current_loss + self.min_delta < self.best_loss:
            self.best_loss = current_loss
            self.counter = 0
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.should_stop = True

class ModelTrainer():
    def __init__(self, model: torch.nn.Module, model_name: str, save_path: str,
kfold_loader: KFoldSpecsDataLoader, lr: float, epochs: int, l2_reg: float):
        """
        :param model: Model to train.
        :param model_name: Name of the model (specstr, pure_specstr and e.t.c.).
        :param save_path: Path to save checkpoints and trained model.
        :param kfold_loader: Loader of train data. Iterable object with train/val
loaders as elements.
        :param lr: learning rate.
        :param epochs: Number of train epochs.
        :param l2_reg: Regularization for optimizer.
        """
```

```
        self.model = model
        self.model_name = model_name
        self.save_path = save_path
        self.kfold_loader = kfold_loader
        self.epochs = epochs

        self.fold = 0
        self.epoch = 0
        self.iteration = 0
        self.best_vloss = float('inf')
        self.folds = len(kfold_loader)
        self.report_times = 20
        self.l2_reg = l2_reg
        self.lr = lr

        self.cuda_scaler = torch.amp.GradScaler("cuda")

        self.start_timestamp = None
        self.timestamp = None
        self.date = None
        self.writer = None

        self.early_stopper = EarlyStopping(patience=7, min_delta=1e-4)
        self.sheduler_one_cycle = None
        self.sheduler_on_plateau = None

    def init_new_train(self):
        # Initialize writers, timestamps.
        self.start_timestamp = datetime.now()
        self.date = self.start_timestamp.strftime(DATE_FORMAT)
        self.timestamp = self.start_timestamp.strftime(TIMESTAMP_FORMAT)
        self.writer = SummaryWriter(f'runs/train_{self.model_name}_{self.timestamp}')

    def init_continue_train(self, saved_model_name):
        saved_model_path = os.path.join(self.save_path, saved_model_name)

        if not os.path.isfile(saved_model_path):
            raise FileNotFoundError(f"Not found model {saved_model_name} by path
{self.save_path}!")

        if "checkpoint" in saved_model_name:
            ckpt = torch.load(saved_model_path, map_location=self.model.device)
            self.model.load_state_dict(ckpt["model_state_dict"])
            self._recreate_optimizer_and_shedulers(1)

            self.optimizer.load_state_dict(ckpt["optimizer_state_dict"])
            self.sheduler_one_cycle.load_state_dict(ckpt["one_cycle_state_dict"])
            self.sheduler_on_plateau.load_state_dict(ckpt["on_plateau_state_dict"])
            self.iteration = ckpt["iteration"] + 1
            self.epoch = ckpt["epoch"] + 1
            self.fold = ckpt["fold"]

            if self.epoch == self.epochs:
                self.fold += 1
                self.epoch = 0

            self.kfold_loader.set_start(self.fold)  # start loading folds from check-
point's fold
```

```
            match = re.search(r".*?(\d{6})_(\d{6}).*", saved_model_name)

            if match:
                date_str, time_str = match.groups()
                self.start_timestamp = datetime.strptime(f"{date_str}_{time_str}",
TIMESTAMP_FORMAT)
            else:
                print(f"Invalid checkpoint file name: expected
'{self.model_name}_checkpoint_<{TIMESTAMP_FORMAT}>_fold_<number>_epoch_<number>',
found: {saved_model_name}")
                self.start_timestamp = datetime.now()
        else:
            self.model.load_state_dict(torch.load(saved_model_path,
weights_only=True))
            self.start_timestamp = datetime.now()

        self.date = self.start_timestamp.strftime(DATE_FORMAT)
        self.timestamp = self.start_timestamp.strftime(TIMESTAMP_FORMAT)
        self.writer = SummaryWriter(f'runs/train_{self.model_name}_{self.timestamp}')
        print(f"Loaded model:\n", self.model)

    def train_model(self):
        if self.start_timestamp is None:
            print("First, call init_new_train()/init_continue_train()!")
            return

        # For every fold.
        for train_loader, val_loader in self.kfold_loader:
            # If epoch is 0 (we start not from checkpoint) then recreate shedulers
and AdamW
            if self.epoch == 0:
                self._recreate_optimizer_and_shedulers(len(train_loader))

            # And for every epoch.
            while self.epoch < self.epochs:
                print(f"Fold {self.fold + 1}/{self.folds}; Epoch {self.epoch +
1}/{self.epochs}")

                # Train for one epoch.
                start_time = time()
                train_avg_loss = self._train_one_epoch(train_loader)
                epoch_train_time = time() - start_time

                # Validate model.
                start_time = time()
                val_avg_loss = self._validate_one_epoch(val_loader)
                epoch_val_time = time() - start_time

                self.sheduler_on_plateau.step(val_avg_loss)     # reduce lr if no
improvement
                self.writer.add_scalar('Loss/validation', val_avg_loss,
self.iteration)

                # Remember best validation loss.
                if val_avg_loss < self.best_vloss:
                    self.best_vloss = val_avg_loss
```

```
                    # Save best in fold model.
                    torch.save({
                        'fold': self.fold,
                        'epoch': self.epoch,
                        'iteration': self.iteration,
                        'model_state_dict': self.model.state_dict(),
                        'optimizer_state_dict': self.optimizer.state_dict(),
                        'one_cycle_state_dict': self.sheduler_one_cycle.state_dict(),
                        'on_plateau_state_dict':
self.sheduler_on_plateau.state_dict()
                    }, os.path.join(self.save_path,
f"{self.model_name}_checkpoint_{self.timestamp}_fold_{self.fold +
1}_epoch_{self.epoch + 1}.pth"))


                current_lr = self.optimizer.param_groups[0]['lr']
                self.writer.add_scalar('Learning rate', current_lr, self.iteration)


                # Log loss and metrics.
                print(f"\n Fold {self.fold + 1}; Epoch {self.epoch + 1} - Training
loss: {train_avg_loss:.3f}; Validation loss: {val_avg_loss:.3f}; lr: {cur-
rent_lr:.2e}")
                print(f"Train time: {epoch_train_time:.3f}; validation time:
{epoch_val_time:.3f}, total epoch time: {(epoch_train_time + epoch_val_time):.3f}\n")

                torch.cuda.empty_cache()
                self.iteration += 1

                # Early stopping.
                self.early_stopper.step(val_avg_loss)
                if self.early_stopper.should_stop:
                    print(f"Early stopping: no improvement for
{self.early_stopper.patience} epochs.")
                    break

                self.epoch += 1

            self.fold += 1
            self.epoch = 0
            self.best_vloss = float('inf')

        # Train end!
        self.writer.close()

        # Get total train time and formate it.
        end_timestamp = datetime.now()
        total_learning_time = (end_timestamp - self.start_timestamp)
        days = total_learning_time.days
        hours, remainder = divmod(total_learning_time.seconds, 3600)
        minutes, seconds = divmod(remainder, 60)
        formated_learning_time = f"{days:02d} days,
{hours:02d}:{minutes:02d}:{seconds:02d}"

        # Close writer and save trained model. Saved model naming is model_name +
moods number + timestamp. Save only weigths.
        model_save_path = os.path.join(self.save_path,
f"{self.model_name}_{end_timestamp.strftime(TIMESTAMP_FORMAT)}.pth")
```

```
        torch.save(self.model.state_dict(), model_save_path)

        print(f"Model saved to {model_save_path}\n\t best validation loss:
{self.best_vloss:.3f}; total learning time: {formated_learning_time}")

    def _recreate_optimizer_and_shedulers(self, epoch_steps) -> None:
        # AdamW optimizer. Use weigth decay and adaptive learning rate.
        self.optimizer = optim.AdamW(self.model.parameters(), lr=self.lr,
weight_decay=self.l2_reg)

        # Sheduler: OneCycleLR (per batch)
        fold_steps = self.epochs * epoch_steps
        self.sheduler_one_cycle = torch.optim.lr_scheduler.OneCycleLR(
            self.optimizer,
            max_lr=self.lr * 10,
            total_steps=fold_steps,
            pct_start=0.15,         # 15% for warm-up
            div_factor=10,          # div factor for start lr
            final_div_factor=1e3,   # div factor for final lr
            anneal_strategy='cos'   # cos strategy for decrease lr
        )

        # Sheduler: ReduceLROnPlateau. Apply after 3 epochs without improving.
        self.sheduler_on_plateau = torch.optim.lr_scheduler.ReduceLROnPlateau(
            self.optimizer,
            mode='min',
            factor=0.2,
            patience=2,
            eps=1e-9,
        )

        # Set early stoping epochs without improving counter to zero.
        self.early_stopper.counter = 0
        self.early_stopper.should_stop = False

    def _train_one_epoch(self, loader):
        pass

    def _validate_one_epoch(self, loader):
        pass


class ClassificationModelTrainer(ModelTrainer):
    """
    Model trainer. Save checkpoints and trained model to save_path.
    Correctly works with two losses and tasks types: multilabel classification, sin-
gle-label classification, autoencoder regression
    """
    def __init__(self, model, model_name: str, save_path: str, target_mode: str,
kfold_loader: KFoldSpecsDataLoader, lr: float, epochs: int, l2_reg: float,
num_classes: int = None):
        """
        :param num_classes: Number of moods to classify. If None -> target_mode
should not be classification.
        :param task_type: Type of the task: multilabel classification, single-label
classification or autoencoder regression.
        """
```

```
        super().__init__(model, model_name, save_path, kfold_loader, lr, epochs,
l2_reg)
        self.target_mode = target_mode

        # Select metrics, loss by type of classification task.
        if target_mode == ONE_HOT_TARGET:
            self.precision_metric = MulticlassPrecision(num_classes=num_classes, av-
erage='macro').to(model.device)
            self.recall_metric = MulticlassRecall(num_classes=num_classes, aver-
age='macro').to(model.device)
            self.f1_metric = MulticlassF1Score(num_classes=num_classes, aver-
age='macro').to(model.device)

            self.loss_function = torch.nn.CrossEntropyLoss(label_smoothing=0.1)
            self.is_multilabel = False
        elif target_mode == MULTILABEL_TARGET:
            self.precision_metric = MultilabelPrecision(num_labels=num_classes, aver-
age='micro').to(model.device)
            self.recall_metric = MultilabelRecall(num_labels=num_classes, aver-
age='micro').to(model.device)
            self.f1_metric = MultilabelF1Score(num_labels=num_classes, aver-
age='micro').to(model.device)

            self.loss_function = torch.nn.BCEWithLogitsLoss()
            self.is_multilabel = True
        elif target_mode == AUTOENCODER_TARGET:
            self.loss_function = torch.nn.MSELoss()
        else:
            raise ValueError(f"Unknown target mode provided: {target_mode}")

    def _compute_and_reset_metrics(self):
        precision = self.precision_metric.compute().item()
        recall = self.recall_metric.compute().item()
        f1 = self.f1_metric.compute().item()

        self.precision_metric.reset()
        self.recall_metric.reset()
        self.f1_metric.reset()

        return precision, recall, f1

    def _train_one_epoch(self, loader):
        total_batches = len(loader)
        report_interval = max(1, total_batches // self.report_times)
        self.model.train(True)
        running_loss = 0.
        avg_loss = 0.

        start_time = time()

        for i, data in enumerate(loader):
            inputs, labels = data
            audio, specs = inputs
            audio = audio.to(self.model.device, non_blocking=True)
            specs = specs.to(self.model.device, non_blocking=True)
            # inputs = inputs.to(self.model.device, non_blocking=True)
            labels = labels.to(self.model.device, non_blocking=True).long()
```

```
            self.optimizer.zero_grad()

            with torch.amp.autocast("cuda"):
                outputs = self.model(audio, specs)
                loss = self.loss_function(outputs, labels)

            # Scaled Backward Pass and gradient Clipping
            self.cuda_scaler.scale(loss).backward()
            self.cuda_scaler.unscale_(self.optimizer)
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)

            self.cuda_scaler.step(self.optimizer)
            self.cuda_scaler.update()

            # Use lr sheduler.
            self.sheduler_one_cycle.step()

            running_loss += loss.item()

            self.update_classification_metrics(outputs, labels)

            # Report 20 times per epoch
            if i % report_interval == report_interval - 1:
                time_per_batch = (time() - start_time) / report_interval
                avg_loss = running_loss / report_interval

                print(f'\t batch [{i + 1}/{total_batches}] - loss: {avg_loss:.5f}\t
time per batch: {time_per_batch:.2f}')
                current_step = self.iteration * total_batches + i
                self.writer.add_scalar('Loss/train', avg_loss, current_step)
                running_loss = 0.
                start_time = time()

        # Log metrics.
        precision, recall, f1 = self._compute_and_reset_metrics()

        self.writer.add_scalar('Precision/train', precision, self.iteration)
        self.writer.add_scalar('Recall/train', recall, self.iteration)
        self.writer.add_scalar('F1/train', f1, self.iteration)

        print(f"\t Training: precision: {precision:.3f}\t recall: {recall:.3f}\t F1:
{f1:.3f}\n")
        return avg_loss

    def _validate_one_epoch(self, loader):
        self.model.eval()  # Set the model to evaluation mode
        val_batches = len(loader)
        running_loss = 0.

        with torch.no_grad():
            for i, data in enumerate(loader):
                inputs, labels = data

                audio, specs = inputs
                audio = audio.to(self.model.device, non_blocking=True)
                specs = specs.to(self.model.device, non_blocking=True)

                # inputs = inputs.to(self.model.device, non_blocking=True)
```

```
                labels = labels.to(self.model.device, non_blocking=True).long()

                outputs = self.model(audio, specs)
                loss = self.loss_function(outputs, labels)

                running_loss += loss
                self.update_classification_metrics(outputs, labels)

        val_avg_loss = running_loss / val_batches

        # Log metrics.
        precision, recall, f1 = self._compute_and_reset_metrics()

        self.writer.add_scalar('Precision/validation', precision, self.iteration)
        self.writer.add_scalar('Recall/validation', recall, self.iteration)
        self.writer.add_scalar('F1/validation', f1, self.iteration)

        print(f"\t Validation: precision: {precision:.3f}\t recall: {recall:.3f}\t
F1: {f1:.3f}\n")
        return val_avg_loss


    def update_classification_metrics(self, model_output, labels):
        if self.is_multilabel:
            labels_true = labels.int()
            labels_pred = (model_output > 0.5).int()
        else:
            labels_true = labels.long()
            labels_pred = torch.argmax(model_output, dim=1)

        self.precision_metric.update(labels_pred, labels_true)
        self.recall_metric.update(labels_pred, labels_true)
        self.f1_metric.update(labels_pred, labels_true)


class AutoencoderModelTrainer(ModelTrainer):
    """
    Model trainer. Save checkpoints and trained model to save_path.
    Correctly works with autoencoder regression
    """
    def __init__(self, model, model_name: str, save_path: str, kfold_loader: KFold-
SpecsDataLoader, lr: float, epochs: int, l2_reg: float):
        super().__init__(model, model_name, save_path, kfold_loader, lr, epochs,
l2_reg)
        self.loss_function = torch.nn.MSELoss()

    def _train_one_epoch(self, loader):
        total_batches = len(loader)
        report_interval = max(1, total_batches // self.report_times)
        self.model.train(True)
        running_loss = 0.
        avg_loss = 0.

        start_time = time()

        for i, data in enumerate(loader):
            inputs, _ = data
            inputs = inputs.to(self.model.device, non_blocking=True)
```

```
            self.optimizer.zero_grad()

            with torch.amp.autocast("cuda"):
                outputs = self.model(inputs)
                loss = self.loss_function(inputs, outputs)

            # Scaled Backward Pass and gradient Clipping
            self.cuda_scaler.scale(loss).backward()
            self.cuda_scaler.unscale_(self.optimizer)
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)

            self.cuda_scaler.step(self.optimizer)
            self.cuda_scaler.update()

            # Use lr sheduler.
            self.sheduler_one_cycle.step()

            running_loss += loss.item()

            # Report 20 times per epoch
            if i % report_interval == report_interval - 1:
                time_per_batch = (time() - start_time) / report_interval
                avg_loss = running_loss / report_interval

                print(f'\t batch [{i + 1}/{total_batches}] - loss: {avg_loss:.5f}\t
time per batch: {time_per_batch:.2f}')
                current_step = self.iteration * total_batches + i
                self.writer.add_scalar('Loss/train', avg_loss, current_step)
                running_loss = 0.
                start_time = time()

        return avg_loss

    def _validate_one_epoch(self, loader):
        self.model.eval()  # Set the model to evaluation mode
        val_batches = len(loader)
        running_loss = 0.

        with torch.no_grad():
            for i, data in enumerate(loader):
                inputs, _ = data
                inputs = inputs.to(self.model.device, non_blocking=True)

                outputs = self.model(inputs)
                loss = self.loss_function(inputs, outputs)

                running_loss += loss

        val_avg_loss = running_loss / val_batches
        return val_avg_loss


def evaluate_classification_model(model, classes, target_mode, test_loader):
    num_classes = len(classes)
    if target_mode == ONE_HOT_TARGET:
        precision_metric = MulticlassPrecision(num_classes=num_classes, aver-
age='macro').to(model.device)
```

```
        recall_metric = MulticlassRecall(num_classes=num_classes, aver-
age='macro').to(model.device)
        f1_metric = MulticlassF1Score(num_classes=num_classes, aver-
age='macro').to(model.device)

        loss_function = torch.nn.CrossEntropyLoss()
        is_multilabel = False
    elif target_mode == MULTILABEL_TARGET:
        precision_metric = MultilabelPrecision(num_labels=num_classes, aver-
age='micro').to(model.device)
        recall_metric = MultilabelRecall(num_labels=num_classes, aver-
age='micro').to(model.device)
        f1_metric = MultilabelF1Score(num_labels=num_classes, aver-
age='micro').to(model.device)

        loss_function = torch.nn.BCEWithLogitsLoss()
        is_multilabel = True
    else:
        raise ValueError(f"Unknown target mode provided: {target_mode}")

    # For confusion matrix.
    all_preds = []
    all_labels = []

    print("Evaluating model...")
    model.eval()  # Set the model to evaluation mode
    running_loss = 0.
    start_time = time()

    # Testing.
    with torch.no_grad():
        for i, data in enumerate(test_loader):
            inputs, labels = data

            audio, specs = inputs
            audio = audio.to(model.device, non_blocking=True)
            specs = specs.to(model.device, non_blocking=True)
            # inputs = inputs.to(model.device, non_blocking=True)
            labels = labels.to(model.device, non_blocking=True).long().squeeze()

            outputs = model(audio, specs)
            loss = loss_function(outputs, labels)

            running_loss += loss

            if is_multilabel:
                labels_true = labels.int()
                labels_pred = (outputs > 0.5).int()
            else:
                labels_true = labels.long()
                labels_pred = torch.argmax(outputs, dim=1)

            precision_metric.update(labels_pred, labels_true)
            recall_metric.update(labels_pred, labels_true)
            f1_metric.update(labels_pred, labels_true)

            all_preds.extend(labels_pred.cpu().numpy())
            all_labels.extend(labels_true.cpu().numpy())
```

```
    test_avg_loss = running_loss / len(test_loader)
    test_time = time() - start_time

    # Log loss and time.
    print(f"Test time: {test_time:.3f}\t loss: {test_avg_loss:.3f}")

    # Compute and write remembered test metrics.
    precision = precision_metric.compute().item()
    recall = recall_metric.compute().item()
    f1 = f1_metric.compute().item()

    # Log metrics.
    print(f"Test precision: {precision:.3f}\t recall: {recall:.3f}\t F1: {f1:.3f}")

    # Build confusion matrix.
    cm = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=classes, yticklabels=classes)
    plt.xlabel("Predicted label")
    plt.ylabel("True label")
    plt.title("Confusion Matrix")
    plt.show()



def evaluate_autoencoder(model, test_loader):
    print("Evaluating model...")
    model.eval()  # Set the model to evaluation mode
    running_loss = 0.
    start_time = time()
    loss_function = torch.nn.MSELoss()

    # Testing.
    with torch.no_grad():
        for i, data in enumerate(test_loader):
            inputs, _ = data
            inputs = inputs.to(model.device, non_blocking=True)

            outputs = model(inputs)
            loss = loss_function(inputs, outputs)

            running_loss += loss

    test_avg_loss = running_loss / len(test_loader)
    test_time = time() - start_time

    # Log loss and time.
    print(f"Test time: {test_time:.3f}\t loss: {test_avg_loss:.3f}")


def get_params_count(model: torch.nn.Module) -> tuple:
    """
    Get number of parameters in the model.
    :return: tuple (total_params, trainable_params)
    """
    total_params = sum(p.numel() for p in model.parameters())
```

```
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    return total_params, trainable_params


def train_text_sentiment_model(
    model: torch.nn.Module,
    model_name: str,
    save_path: str,
    num_classes: int,
    train_loader,
    val_loader,
    lr: float,
    epochs: int,
    l2_reg: float,
):
    """
    Train text sentiment analysis transformer model.
    """

    start_timestamp = datetime.now()
    timestamp = start_timestamp.strftime(TIMESTAMP_FORMAT)
    writer = SummaryWriter(f'runs/train_{model_name}_{timestamp}')
    report_interval = 5

    total_batches = len(train_loader)

    # AdamW optimizer. Use weigth decay and adaptive learning rate.
    optimizer = optim.AdamW(model.parameters(), lr=lr, weight_decay=l2_reg)

    precision_metric = MulticlassPrecision(num_classes=num_classes, aver-
age='macro').to(model.device)
    recall_metric = MulticlassRecall(num_classes=num_classes, aver-
age='macro').to(model.device)
    f1_metric = MulticlassF1Score(num_classes=num_classes, aver-
age='macro').to(model.device)

    loss_function = torch.nn.CrossEntropyLoss()
    cuda_scaler = torch.amp.GradScaler("cuda")

    # from transformers import WhisperTokenizer
    # tokenizer = WhisperTokenizer.from_pretrained("openai/whisper-small")

    for epoch in range(epochs):
        train_total_loss = 0.
        running_loss = 0.0
        start_time = time()
        model.train()

        print(f"\n Epoch {epoch + 1}")

        for i, batch in enumerate(train_loader):
            # print(tokenizer.decode(batch["input_ids"][0],
skip_special_tokens=True))
            # print(batch["labels"][0])

            input_ids = batch["input_ids"].to(model.device, non_blocking=True)
            attention_mask = batch["attention_mask"].to(model.device,
non_blocking=True)
```

```
            labels = batch["labels"].to(model.device, non_blocking=True).long()

            with torch.amp.autocast("cuda"):
                outputs = model(input_ids, attention_mask)
                loss = loss_function(outputs, labels)

            # Scaled Backward Pass and gradient Clipping
            cuda_scaler.scale(loss).backward()
            cuda_scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

            cuda_scaler.step(optimizer)
            cuda_scaler.update()

            running_loss += loss.item()

            labels_pred = torch.argmax(outputs, dim=1)

            precision_metric.update(labels_pred, labels)
            recall_metric.update(labels_pred, labels)
            f1_metric.update(labels_pred, labels)

            # Report 20 times per epoch
            if i % report_interval == report_interval - 1:
                time_per_batch = (time() - start_time) / report_interval
                avg_loss = running_loss / report_interval
                train_total_loss += avg_loss

                print(f'\t batch [{i + 1}/{total_batches}] - loss: {avg_loss:.5f}\t
time per batch: {time_per_batch:.2f}')
                current_step = epoch * total_batches + i
                writer.add_scalar('Loss/train', avg_loss, current_step)
                running_loss = 0.
                start_time = time()

        # Log metrics.
        precision = precision_metric.compute().item()
        recall = recall_metric.compute().item()
        f1 = f1_metric.compute().item()

        precision_metric.reset()
        recall_metric.reset()
        f1_metric.reset()

        writer.add_scalar('Precision/train', precision, epoch)
        writer.add_scalar('Recall/train', recall, epoch)
        writer.add_scalar('F1/train', f1, epoch)

        print(f"\t Training: precision: {precision:.3f}\t recall: {recall:.3f}\t F1:
{f1:.3f}\n")

        # Validation
        model.eval()
        val_batches = len(val_loader)
        running_loss = 0.

        with torch.no_grad():
            for i, batch in enumerate(val_loader):
```

```
                input_ids = batch["input_ids"].to(model.device, non_blocking=True)
                attention_mask = batch["attention_mask"].to(model.device,
non_blocking=True)
                labels = batch["labels"].to(model.device, non_blocking=True).long()

                outputs = model(input_ids, attention_mask)
                loss = loss_function(outputs, labels)

                running_loss += loss

                labels_pred = torch.argmax(outputs, dim=1)

                precision_metric.update(labels_pred, labels)
                recall_metric.update(labels_pred, labels)
                f1_metric.update(labels_pred, labels)

        val_avg_loss = running_loss / val_batches

        writer.add_scalar('Loss/validation', val_avg_loss, epoch)

        # Log metrics.
        precision = precision_metric.compute().item()
        recall = recall_metric.compute().item()
        f1 = f1_metric.compute().item()

        precision_metric.reset()
        recall_metric.reset()
        f1_metric.reset()

        writer.add_scalar('Precision/validation', precision, epoch)
        writer.add_scalar('Recall/validation', recall, epoch)
        writer.add_scalar('F1/validation', f1, epoch)

        print(f"\t Epoch: {epoch + 1}\t Train average loss: {train_total_loss / to-
tal_batches:.3f}\t Validation average loss: {val_avg_loss:.3f}\n")
        print(f"\t Validation: precision: {precision:.3f}\t recall: {recall:.3f}\t
F1: {f1:.3f}\n")

        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
        }, os.path.join(save_path, f"{model_name}_checkpoint_{timestamp}_epoch_{epoch
+ 1}.pth"))

    # Train end!
    writer.close()

    # Get total train time and formate it.
    end_timestamp = datetime.now()
    total_learning_time = (end_timestamp - start_timestamp)
    days = total_learning_time.days
    hours, remainder = divmod(total_learning_time.seconds, 3600)
    minutes, seconds = divmod(remainder, 60)
    formated_learning_time = f"{days:02d} days,
{hours:02d}:{minutes:02d}:{seconds:02d}"
```

```
    # Close writer and save trained model. Saved model naming is model_name + moods
number + timestamp. Save only weigths.
    model_save_path = os.path.join(save_path,
f"{model_name}_{end_timestamp.strftime(TIMESTAMP_FORMAT)}.pth")
    torch.save(model.state_dict(), model_save_path)

    print(f"Model saved to {model_save_path}\n\t total learning time: {for-
mated_learning_time}")
```

# ПРИЛОЖЕНИЕ Г – Листинг архитектур моделей

```
import torch
from torch import nn
from model.layer import *


class SpectrogramPureTransformer(nn.Module):
    def __init__(self, output_dim: int, device='cuda'):
        super().__init__()
        self.device = device
        D_MODEL = 96
        NHEAD = 16
        NUM_LAYERS = 4

        encoder_layer = nn.TransformerEncoderLayer(d_model=D_MODEL, nhead=NHEAD,
batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
num_layers=NUM_LAYERS)

        self.attention_pool = nn.Sequential(
            nn.Linear(D_MODEL, D_MODEL//2),
            nn.Tanh(),
            nn.Linear(D_MODEL//2, 1),
            nn.Softmax(dim=1)
        )

        self.output_proj = nn.Linear(D_MODEL, output_dim)

    def forward(self, x):
        # x = truncate_spec(x, self.seq_len)  # shape (batch, mel_features, seq_len)
        x = x.permute(0, 2, 1)                 # shape (batch, seq_len, mel_features)

        x = self.transformer_encoder(x)

        # Attention pooling
        attn_weights = self.attention_pool(x)  # (batch, seq, 1)
        x = torch.sum(x * attn_weights, dim=1)  # (batch, d_model)

        # Final projection
        logits = self.output_proj(x)
        return logits

    def __str__(self):
        model_describe = ""
        model_describe += str(self.transformer_encoder) + "\n"
        model_describe += str(self.attention_pool) + "\n"
        model_describe += str(self.output_proj) + "\n"
        return model_describe


class SpectrogramTransformer(nn.Module):
    def __init__(self, output_dim: int, dropout=0.2, device='cuda'):
        super().__init__()
        self.device = device

        # Model params
```

```
        CNN_UNITS = [512, 512, 1024, 1024, 512]
        CNN_KERNELS = [3] * len(CNN_UNITS)
        CNN_STRIDES = [2] * len(CNN_UNITS)
        CNN_PADDINGS = [1] * len(CNN_UNITS)
        CNN_RES_CON = [False] * len(CNN_UNITS)
        RNN_UNITS = 256
        RNN_LAYERS = 2
        TRANSFORMER_DEPTH = 432
        NHEAD = 6
        NUM_ENCODERS = 4

        TRANSFORMER_DROPOUT = 0.3

        self.congruformer = ConGRUFormer(
            in_channels=96,
            cnn_units=CNN_UNITS,
            cnn_kernel_sizes=CNN_KERNELS,
            cnn_strides=CNN_STRIDES,
            cnn_paddings=CNN_PADDINGS,
            cnn_res_con=CNN_RES_CON,
            rnn_units=RNN_UNITS,
            rnn_layers=RNN_LAYERS,
            transformer_depth=TRANSFORMER_DEPTH,
            nhead=NHEAD,
            num_encoders=NUM_ENCODERS,
            dropout=dropout,
            transformer_dropout=TRANSFORMER_DROPOUT,
            device=device
        )

        self.output_proj = nn.Linear(TRANSFORMER_DEPTH, output_dim)

    def forward(self, x):
        # CNN Feature extraction
        x = self.congruformer(x)  # (batch, cnn_units, seq)
        return self.output_proj(x)

class SpectrogramSmallTransformer(nn.Module):
    def __init__(self, output_dim: int, dropout=0.2, device='cuda'):
        super().__init__()
        self.device = device

        # Model params
        CNN_UNITS = [128, 256, 512, 1024, 512]
        CNN_KERNELS = [3] * len(CNN_UNITS)
        CNN_STRIDES = [2] * len(CNN_UNITS)
        CNN_PADDINGS = [0] * len(CNN_UNITS)
        CNN_RES_CON = [False] * len(CNN_UNITS)
        RNN_UNITS = 256
        RNN_LAYERS = 2
        TRANSFORMER_DEPTH = 312
        NHEAD = 6
        NUM_ENCODERS = 4

        TRANSFORMER_DROPOUT = 0.3

        self.congruformer = ConGRUFormer(
            in_channels=96,
```

```
            cnn_units=CNN_UNITS,
            cnn_kernel_sizes=CNN_KERNELS,
            cnn_strides=CNN_STRIDES,
            cnn_paddings=CNN_PADDINGS,
            cnn_res_con=CNN_RES_CON,
            rnn_units=RNN_UNITS,
            rnn_layers=RNN_LAYERS,
            transformer_depth=TRANSFORMER_DEPTH,
            nhead=NHEAD,
            num_encoders=NUM_ENCODERS,
            dropout=dropout,
            transformer_dropout=TRANSFORMER_DROPOUT,
            device=device
        )

        self.output_proj = nn.Linear(TRANSFORMER_DEPTH, output_dim)

    def forward(self, x):
        # CNN Feature extraction
        x = self.congruformer(x)  # (batch, cnn_units, seq)
        return self.output_proj(x)

class SpectrogramMaskedAutoEncoder(nn.Module):
    def __init__(self, mask_ratio=0.8, dropout=0.2, device='cuda'):
        super().__init__()
        self.mask_ratio = mask_ratio
        self.device = device

        # Model params
        INPUT_CHANNELS = 96
        KERNEL_SIZE = 2
        PADDING = 1

        TRANSFORMER_DEPTH = 256
        NHEAD = 8
        NUM_ENCODER_LAYERS = 6
        NUM_DECODER_LAYERS = 4
        TRANSFORMER_CONTEXT_LEN = 128

        DROPOUT_CNN = 0.1
        DROPOUT_TRANSFORMER = 0.4

        # --- Encoder ---
        self.encoder = SpectrogramMaskedEncoder(INPUT_CHANNELS, KERNEL_SIZE, PADDING,
TRANSFORMER_DEPTH, TRANSFORMER_CONTEXT_LEN,
                                                NHEAD, NUM_ENCODER_LAYERS,
mask_ratio=mask_ratio, dropout_cnn=DROPOUT_CNN,
                                                drop-
out_transformer=DROPOUT_TRANSFORMER, device=device)

        # --- Decoder ---
        self.decoder = SpectrogramMaskedDecoder(INPUT_CHANNELS, KERNEL_SIZE, PADDING,
TRANSFORMER_DEPTH, NHEAD, NUM_DECODER_LAYERS,
                                                DROPOUT_CNN, DROPOUT_TRANSFORMER,
device=device)

    def forward(self, spec):
        """
```

```
        spec: Tensor (batch, features, sequence)
        returns: recon_spec (batch, features, sequence)
        """
        initial_len = spec.size(-1)
        encoded_vis_spec, masked_full_spec, mask = self.encoder(spec)       # remem-
ber mask also
        decoded_spec = self.decoder(masked_full_spec, encoded_vis_spec)     # decoded
spec length may differ

        return nn.functional.interpolate(decoded_spec, size=initial_len,
mode='linear')


class SpectrogramMaskedEncoder(nn.Module):
    def __init__(self, input_channels, kernel_size, padding, transformer_depth,
transformer_context_len,
                 nhead, num_encoder_layers, mask_ratio=0.8, dropout_cnn=0.1, drop-
out_transformer=0.4, device='cuda'):
        super().__init__()
        self.device = device
        self.nhead = nhead
        self.transformer_depth = transformer_depth
        self.mask_ratio = mask_ratio

        self.cnn_encoder = nn.Sequential(
            nn.Conv1d(in_channels=input_channels, out_channels=128, ker-
nel_size=kernel_size, padding=padding, bias=False),
            nn.BatchNorm1d(128),
            nn.GELU(),
            nn.MaxPool1d(kernel_size=kernel_size),
            nn.Dropout1d(dropout_cnn),

            # nn.Conv1d(in_channels=128, out_channels=256, kernel_size=kernel_size,
padding=padding, bias=False),
            # nn.BatchNorm1d(256),
            # nn.GELU(),
            # nn.MaxPool1d(kernel_size=kernel_size),
            # nn.Dropout1d(dropout_cnn),

            nn.Conv1d(in_channels=128, out_channels=transformer_depth, ker-
nel_size=kernel_size, padding=padding, bias=False),
            nn.BatchNorm1d(transformer_depth),
            nn.GELU(),
            nn.AdaptiveMaxPool1d(output_size=transformer_context_len),
            nn.Dropout1d(dropout_cnn),
        )
        # Output of CNN is (batch, d_model, transformer_context_len)

        # Use sinusoidal positional encoding
        self.pos_encoder = PositionalEncoding(transformer_depth)
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=transformer_depth,
            dim_feedforward=transformer_depth * 4,
            nhead=nhead,
            dropout=dropout_transformer,
            activation='gelu',
            batch_first=True,
            norm_first=True     # LayerNorm first
```

```
        )
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
num_layers=num_encoder_layers, enable_nested_tensor=False)

        # Trainable mask token (to avoid masking with padding value)
        self.mask_token = nn.Parameter(torch.zeros(transformer_depth))

    def forward(self, x, gen_mask=True):
        # CNN specs encoding. Seq len decreasing
        x = self.cnn_encoder(x)  # (batch, cnn_units, seq)
        x = x.permute(0, 2, 1)  # (batch, seq, cnn_units)

        # Generate random mask if required.
        if gen_mask:
            seq_len = x.size(1)

            # Generate mask (single mask for all of batches)
            num_vis = int(round(seq_len * (1 - self.mask_ratio)))

            perm = torch.randperm(seq_len, device=self.device)
            vis_idx  = perm[:num_vis].argsort(dim=0)

            mask = torch.zeros(seq_len, dtype=torch.bool, device=self.device)
            mask[vis_idx] = True

            # Apply mask for BxS along whole depth. Get unmasked and masked X.
            x_vis = x[:, vis_idx, :]  # encode only unmasked parts.

            x_masked = torch.empty_like(x)
            i_x_vis = 0
            for i in range(len(mask)):
                if mask[i]:
                    x_masked[:, i, :] = x_vis[:, i_x_vis, :]
                    i_x_vis += 1
                else:
                    x_masked[:, i, :] = self.mask_token
        else:
            # This case intended for encoder-only architecture (pre-learned, not au-
toencoder)
            x_vis = x  # if mask not required, encode whole spectrogram.
            x_masked = None
            mask = None

        # Add positional encoding
        x_enc = self.pos_encoder(x_vis)

        # Transformer processing
        x_enc = self.transformer_encoder(x_enc)  # (batch, vis_seq, d_model)

        return x_enc, x_masked, mask

    def __str__(self):
        model_describe = ""
        model_describe += "CNN Encoder: " + str(self.cnn_encoder) + "\n" * 2
        model_describe += str(self.pos_encoder) + "\n" * 2
        model_describe += "Transformer Encoder: " + str(self.transformer_encoder) +
"\n"
        return model_describe
```

```
class SpectrogramMaskedDecoder(nn.Module):
    def __init__(self, input_channels, kernel_size, padding, transformer_depth,
nhead,
                 num_decoder_layers, dropout_cnn=0.1, dropout_transformer=0.4, de-
vice='cuda'):
        super().__init__()
        self.device = device

        decoder_layer = nn.TransformerDecoderLayer(
            d_model=transformer_depth,
            nhead=nhead,
            dropout=dropout_transformer,
            activation='gelu',
            batch_first=True,
            norm_first=True
        )
        self.transformer_decoder = nn.TransformerDecoder(decoder_layer,
num_layers=num_decoder_layers)

        # Output of CNN decoder is (batch, IN_CHANNELS, IN_SEQ_LEN)
        self.cnn_decoder = nn.Sequential(
            nn.ConvTranspose1d(in_channels=transformer_depth,
                               out_channels=128,
                               kernel_size=kernel_size,
                               stride=kernel_size,
                               padding=padding,
                               bias=False),
            nn.BatchNorm1d(128),
            nn.GELU(),
            nn.Dropout1d(dropout_cnn),

            nn.ConvTranspose1d(in_channels=128,
                               out_channels=input_channels,
                               kernel_size=kernel_size,
                               stride=kernel_size,
                               padding=padding,
                               bias=False),
            nn.BatchNorm1d(input_channels),
            nn.GELU(),
            nn.Dropout1d(dropout_cnn),
        )

    def forward(self, dec_in, enc_out):
        # X(dec_in) - decoder masked input (batch, seq_len, d_model);
        # Memory(enc_out) - encoder output (batch, vis_len, d_model).
        x = self.transformer_decoder(dec_in, enc_out)  # (batch, seq, d_model)

        # Final decoding. Increasing of the seq len.
        x = x.permute(0, 2, 1)  # (batch, cnn_units, seq)
        x = self.cnn_decoder(x)

        return x

    def __str__(self):
        model_describe = ""
```

```
        model_describe += "Transformer Decoder: " + str(self.transformer_decoder) +
"\n"
        model_describe += "CNN Decoder: " + str(self.cnn_decoder) + "\n" * 2
        return model_describe


class SpectrogramPreTrainedTransformer(nn.Module):
    def __init__(self, encoder: SpectrogramMaskedEncoder, output_dim: int, drop-
out=0.2, device='cuda'):
        """
        Pre-trained transformer model: encoder -> classifier.
        :param encoder: part of the trained autoencoder.
        :param output_dim: classes to classify number
        """
        super().__init__()
        self.device = device

        self.encoder = encoder
        depth = encoder.transformer_depth
        nhead = encoder.nhead

        self.attention_pool = MultiHeadPool(depth, nhead, dropout=dropout)
        self.output_proj = nn.Linear(depth, output_dim)

    def forward(self, x):
        x, _, _ = self.encoder(x, gen_mask=False)

        # Attention pooling
        x = self.attention_pool(x)  # (batch, depth)

        # Final projection (classification)
        logits = self.output_proj(x)
        return logits

    def __str__(self):
        model_describe = ""
        model_describe += "Encoder: " + str(self.encoder) + "\n"
        model_describe += "Attention pooling: " + str(self.attention_pool) + "\n" * 2
        model_describe += "Output projection: " + str(self.output_proj) + "\n" * 2
        return model_describe

class RawAudioTransformer(nn.Module):
    def __init__(self, output_channels: int, dropout=0.2, device='cuda'):
        super().__init__()
        self.device = device

        # Model params
        CNN_UNITS = [128, 256, 512, 1024, 512, 256]
        CNN_KERNELS = [10, 5, 3, 3, 3, 3]
        CNN_STRIDES = [7, 3, 2, 2, 2, 2]
        CNN_RES_CON = [False, False, True, True, True, True]
        CNN_PADDINGS = [0] * len(CNN_UNITS)
        RNN_UNITS = 256
        RNN_LAYERS = 2
        TRANSFORMER_DEPTH = 312
        NHEAD = 6
        NUM_ENCODERS = 4
```

```
        TRANSFORMER_DROPOUT = 0.3

        self.congruformer = ConGRUFormer(
            in_channels=1,
            cnn_units=CNN_UNITS,
            cnn_kernel_sizes=CNN_KERNELS,
            cnn_strides=CNN_STRIDES,
            cnn_paddings=CNN_PADDINGS,
            cnn_res_con=CNN_RES_CON,
            rnn_units=RNN_UNITS,
            rnn_layers=RNN_LAYERS,
            transformer_depth=TRANSFORMER_DEPTH,
            nhead=NHEAD,
            num_encoders=NUM_ENCODERS,
            dropout=dropout,
            transformer_dropout=TRANSFORMER_DROPOUT,
            device=device
        )

        self.output_proj = nn.Linear(TRANSFORMER_DEPTH, output_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.congruformer(x)
        return self.output_proj(x)


class PretrainedRawAudioTransformer(nn.Module):
    def __init__(self, output_channels: int, dropout=0.2, device='cuda'):
        super().__init__()
        self.device = device

        self.wav2vec = Wav2Vec2Model.from_pretrained(
            "facebook/wav2vec2-base-960h",
            output_hidden_states=True
        )

        # Frozen parameters better and faster training
        for param in self.wav2vec.parameters():
            param.requires_grad = False

        hidden_size = self.wav2vec.config.hidden_size

        self.attention_pooling = MultiHeadPool(
            d_model=hidden_size,
            nhead=8,
            dropout=dropout
        )

        self.output_proj = nn.Linear(hidden_size, output_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        features = self.wav2vec(x.squeeze(1)).last_hidden_state
        pooled = self.attention_pooling(features)
        return self.output_proj(pooled)

class TextTransformer(nn.Module):
    """
    Transformer with positional sinusoidal encoding and attention pooling.
```

```
    """
    def __init__(self, depth: int, nheads: int, num_encoders: int, dropout=0.2, whis-
per_model_name: str = "openai/whisper-small", device='cuda'):
        super().__init__()
        self.depth = depth
        self.device = device

        self.tokenizer = WhisperTokenizer.from_pretrained(whisper_model_name)
        vocab_size = self.tokenizer.vocab_size +
len(self.tokenizer.added_tokens_encoder)

        self.embedding = nn.Embedding(num_embeddings=vocab_size, embedding_dim=depth)

        for param in self.embedding.parameters():
            param.requires_grad = False

        self.pos_encoding = PositionalEncoding(d_model=depth)
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=depth,
            dim_feedforward=depth * 4,
            nhead=nheads,
            dropout=dropout,
            activation='gelu',
            batch_first=True,
            norm_first=True     # LayerNorm first
        )

        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
num_layers=num_encoders)
        self.attention_pool = MultiHeadPool(depth, nheads, dropout)

    def forward(self, tokens_ids: torch.Tensor, padding_mask: torch.Tensor) ->
torch.Tensor:
        """
        tokens_ids: indices of tokens in the text sequence (batch_size, seq_len)
        padding_mask: mask for the text sequence (batch_size, seq_len): 1 for true
tokens, 0 for paddings/special tokens
        """
        x = self.embedding(tokens_ids)  # (batch_size, seq_len, depth)
        x = self.pos_encoding(x)
        x = self.transformer_encoder(x, src_key_padding_mask=~padding_mask.bool())
        x = self.attention_pool(x)
        return x


class TextExtractor(nn.Module):
    def __init__(self, max_seq_len=128, whisper_model_name: str = "openai/whisper-
small", device='cuda'):
        super().__init__()
        self.device = device
        self.max_seq_len = max_seq_len

        # Model params
        self.processor = WhisperProcessor.from_pretrained(whisper_model_name)
        self.model = WhisperForConditionalGenera-
tion.from_pretrained(whisper_model_name)

        for param in self.model.parameters():
```

```
            param.requires_grad = False

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        x: raw mono audio data (batch_size, seq_len) with values between ([-1.0,
+1.0]), 16 kHz.
        Returns:
          - generated_ids: (batch_size, text_len) — generated text token indexes.
          - padding_mask: torch.LongTensor (batch_size, text_len) — mask with zero in
padding/special tokens.
        """
        x = x.squeeze(1).to("cpu")

        list_of_x_els = [x_i for x_i in x.cpu().numpy()]

        p = self.processor(list_of_x_els, sampling_rate=16000, return_tensors="pt")
        input_features = p.input_features.to(self.device)

        generated_ids = self.model.generate(
            input_features,
            max_new_tokens=128,
            eos_token_id=self.processor.tokenizer.eos_token_id,
            pad_token_id=self.processor.tokenizer.pad_token_id,
            use_cache=True,
        )

        pad_id = self.processor.tokenizer.pad_token_id
        padding_mask = (generated_ids != pad_id).long()

        return generated_ids, padding_mask


class TextSentimentTransformer(nn.Module):
    """
    Text sentiment analisys transformer model.
    """
    def __init__(self, output_channels: int, dropout=0.2, whisper_model_name: str =
"openai/whisper-small", device='cuda'):
        super().__init__()
        self.device = device

        # Model params
        TRANSFORMER_DEPTH = 312
        NHEAD = 6
        NUM_ENCODERS = 4

        self.transformer = TextTransformer(TRANSFORMER_DEPTH, NHEAD, NUM_ENCODERS,
dropout, whisper_model_name, device=device)
        self.output_proj = nn.Sequential(
            nn.Linear(TRANSFORMER_DEPTH, TRANSFORMER_DEPTH // 2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(TRANSFORMER_DEPTH // 2, output_channels)
        )

    def forward(self, tokens_ids: torch.Tensor, padding_mask: torch.Tensor) ->
torch.Tensor:
        """
```

```
        tokens_ids: indices of tokens in the text sequence (batch_size, seq_len)
        padding_mask: mask for the text sequence (batch_size, seq_len): 1 for true
tokens, 0 for paddings/special tokens
        """
        x = self.transformer(tokens_ids, padding_mask)
        return self.output_proj(x)


class LirycsSentimentTransformer(nn.Module):
    """
    Predict lirycs sentiment using pipeline: TextExtractor -> TransformerWithPooling
-> Output Projection.
    Input data is raw audio, output is sentiment logits (output_channels,).
    """
    def __init__(self, text_extractor: TextExtractor, transformer: TextTransformer,
device='cuda'):
        super().__init__()
        self.device = device
        self.text_extractor = text_extractor
        self.transformer = transformer
        self.depth = transformer.depth

    def forward(self, waveform: torch.Tensor) -> torch.Tensor:
        text_ids, padding_mask = self.text_extractor(waveform)
        return self.transformer(text_ids, padding_mask)

import torch
from torch import nn

class HeterogeneousDataSentimentClassifier(nn.Module):
    """
    Heterogeneous model that combines different types of models and input data.

    Base of the model - mel-spectrograms model. It is required part of the full mod-
el.
    Additional models can be added for text, audio, and numeric features.
    """
    def __init__(self,
                 output_dim,
                 specs_model: nn.Module,
                 text_model: nn.Module | None = None,
                 audio_model: nn.Module | None = None,
                 dropout: float = 0.2,
                 device='cuda'):
        super().__init__()
        self.specs_model = specs_model

        for param in self.specs_model.parameters():
            param.requires_grad = False

        self.text_model = text_model
        self.audio_model = audio_model

        self.depth = specs_model.depth if hasattr(specs_model, 'depth') else 256

        if text_model is not None:
            self.depth += text_model.depth if hasattr(text_model, 'depth') else 256
```

```
        if audio_model is not None:
            for param in self.audio_model.parameters():
                param.requires_grad = False
            self.depth += audio_model.depth if hasattr(audio_model, 'depth') else 256

        self.output_projection = nn.Sequential(
            nn.Linear(self.depth, 256, bias=False),
            nn.BatchNorm1d(256),
            nn.GELU(),
            nn.Dropout(dropout),

            nn.Linear(256, 128, bias=False),
            nn.BatchNorm1d(128),
            nn.GELU(),
            nn.Dropout(dropout),

            nn.Linear(128, 64, bias=False),
            nn.BatchNorm1d(64),
            nn.GELU(),
            nn.Dropout(dropout),

            nn.Linear(64, output_dim)
        )

        self.device = device

    def forward(self, audio_input, spec_input) -> torch.Tensor:
        spec_features = self.specs_model(spec_input)
        combined_features = spec_features

        if self.text_model is not None:
            text_features = self.text_model(audio_input)            # text input ex-
tracted from raw audio (e.g., Whisper)
            combined_features = torch.cat((combined_features, text_features), dim=1)

        if self.audio_model is not None:
            audio_features = self.audio_model(audio_input)
            combined_features = torch.cat((combined_features, audio_features), dim=1)

        logits = self.output_projection(combined_features)
        return logits

class FeaturesDense(nn.Module):
    def __init__(self, input_channels: 64, output_channels: int, dropout=0.2, de-
vice='cuda'):
        super().__init__()
        self.device = device

        self.mlp = nn.Sequential(
            nn.Linear(input_channels, 128, bias=False),
            nn.BatchNorm1d(128),
            nn.GELU(),
            nn.Dropout(dropout),

            nn.Linear(128, 64, bias=False),
            nn.BatchNorm1d(64),
            nn.GELU(),
            nn.Dropout(dropout),
```

```
        )

        self.output_proj = nn.Linear(64, output_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.mlp(x)
        return self.output_proj(x)

class MultiHeadPool(nn.Module):
    def __init__(self, d_model: int, nhead: int, dropout: float = 0.3):
        """
        Multi-head Attention Pool.
        """
        super().__init__()

        self.query = nn.Parameter(torch.randn(1, d_model))  # learnable query
        self.mha = nn.MultiheadAttention(
            embed_dim=d_model,
            num_heads=nhead,
            batch_first=True,
            dropout=dropout
        )

        self.norm = nn.LayerNorm(d_model)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        batch_size = x.size(0)

        q = self.query.unsqueeze(0).expand(batch_size, -1, -1)  # (batch, 1, depth)

        # out: (batch, 1, depth), attn_weights: (batch, 1, seq_len)
        attn_out, attn_weights = self.mha(q, x, x, need_weights=True)  # K, V = x, x
        out = attn_out.squeeze(1)  # (B, D)

        # Residual connection & normalization
        out = self.norm(out + q.squeeze(1))
        return out


class CustomTransformerEncoderLayer(nn.TransformerEncoderLayer):
    """
    Custom transformer encoder layer with oveloaded forward:
        this encoder applyes Multi-Head Attention with external query q.
    """
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def forward(self, x, q):
        if self.norm_first:
            x = self.norm1(x)

        x, _ = self.self_attn(query=q,
                              key=x,
                              value=x)

        out = q + self.dropout1(x)

        if not self.norm_first:
```

```
            out = self.norm1(out)
        else:
            out = self.norm2(out)

        x = self.linear2(self.dropout(self.activation(self.linear1(out))))

        out = out + self.dropout2(x)

        if not self.norm_first:
            out = self.norm2(out)

        return out

class CustomTransformerEncoder(nn.Module):
    def __init__(self, encoder_layer, num_layers):
        super().__init__()
        self.layers = nn.ModuleList([
            encoder_layer
            for _ in range(num_layers)
        ])

    def forward(self, src, q):
        output = q
        for layer in self.layers:
            output = layer(src, output)
        return output


class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, max_len: int = 5000):
        super().__init__()

        pe = torch.zeros(max_len, d_model)  # positional encoding
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)  # posi-
tions from 0 to max_len-1

        # sinusoidal absolute, wk = 1 / 1000 ^ (2k / d)
        omega = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) /
d_model))

        pe[:, 0::2] = torch.sin(position * omega)  # even code with sin
        pe[:, 1::2] = torch.cos(position * omega)  # odd code with cos

        self.register_buffer('pe', pe)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = x + self.pe[:x.size(1), :].unsqueeze(0)
        return x

    def __str__(self):
        return "Positional sinusoidal encoding"


class ResidualConv1d(nn.Module):
    """
    One convolution with residual connection.
    """
```

```
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, pad-
ding=1):
        super().__init__()
        self.convolution = nn.Sequential(
            nn.Conv1d(in_channels, out_channels, kernel_size, stride, padding, bi-
as=False),
            nn.BatchNorm1d(out_channels),
            nn.GELU()
        )

        # If in_channels != out_channels then apply 1x1 convolution (for identity=x).
        self.channels_projection = None
        if in_channels != out_channels:
            self.channels_projection = nn.Sequential(
                nn.Conv1d(in_channels, out_channels, kernel_size=1, bias=False),
                nn.BatchNorm1d(out_channels)
            )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        out = self.convolution(x)

        # Projection in channels dimension (if required)
        if self.channels_projection is not None:
            x = self.channels_projection(x)

        # Adaptive average pooling in len dimenstion (if required)
        new_len = out.size(2)
        if new_len != x.size(2):
            x = F.adaptive_avg_pool1d(x, new_len)

        # Residual connection
        return out + x

    def __str__(self):
        return f"Residual convolution: {super().__str__()}"

class MultiLayerConv1d(nn.Module):
    def __init__(
            self,
            in_channels: int,
            cnn_units: list,
            cnn_kernel_sizes: list,
            cnn_strides: list,
            cnn_paddings: list,
            residual_connections: list,
            dropout=0.1,
            device='cuda'
    ) -> None:
        """
        Multi-layer 1D convolution.
        """
        super().__init__()
        assert len(cnn_units) == len(cnn_kernel_sizes) == len(cnn_strides) ==
len(cnn_paddings) == len(residual_connections)
        assert len(cnn_units) != 0
        self.device = device

        conv_layers = []
```

```
        current_in = in_channels

        for out_channels, kernel_size, stride, padding, res_con in zip(cnn_units,
cnn_kernel_sizes, cnn_strides, cnn_paddings, residual_connections):
            if res_con:
                conv_layers.extend([
                    ResidualConv1d(
                        in_channels=current_in,
                        out_channels=out_channels,
                        kernel_size=kernel_size,
                        stride=stride,
                        padding=padding
                    ),
                    nn.Dropout(dropout)
                ])
            else:
                conv_layers.extend([
                    nn.Conv1d(
                        in_channels=current_in,
                        out_channels=out_channels,
                        kernel_size=kernel_size,
                        stride=stride,
                        padding=padding,
                        bias=False
                    ),
                    nn.BatchNorm1d(out_channels),
                    nn.GELU(),
                    nn.Dropout(dropout)
                ])

            current_in = out_channels

        self.model = nn.Sequential(*conv_layers)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.model(x)


class ConGRUFormer(nn.Module):
    def __init__(
            self,
            in_channels: int,
            cnn_units: list,
            cnn_kernel_sizes: list,
            cnn_strides: list,
            cnn_paddings: list,
            cnn_res_con: list,
            rnn_units=256,
            rnn_layers=1,
            transformer_depth=256,
            nhead=8,
            num_encoders=6,
            dropout=0.2,
            transformer_dropout=0.3,
            device='cuda'
    ) -> None:
        """
```

```
        Architecture based on 1D convolution, GRU time-sequence processing and Trans-
former processing.
        """
        assert rnn_layers != 0

        super().__init__()
        self.device = device
        self.depth = transformer_depth

        # Time axis compression by CNN
        self.cnn = MultiLayerConv1d(
            in_channels,
            cnn_units=cnn_units,
            cnn_kernel_sizes=cnn_kernel_sizes,
            cnn_strides=cnn_strides,
            cnn_paddings=cnn_paddings,
            residual_connections=cnn_res_con,
            dropout=dropout,
            device=device
        )
        # Output of CNN is (batch, cnn_units[-1], new_sequence_length)
        depth_cnn = cnn_units[-1]

        self.rnn = nn.GRU(depth_cnn, rnn_units, num_layers=rnn_layers,
batch_first=True,
                          dropout=dropout, bidirectional=True)
        depth_rnn = rnn_units * 2

        # Linear projection with layer normalization from CNN/RNN output to TRANS-
FORMER input
        self.x_proj = nn.Sequential(
            nn.Linear(depth_cnn, transformer_depth),
            nn.ReLU()
        )

        self.q_proj = nn.Sequential(
            nn.Linear(depth_rnn, transformer_depth),
            nn.LayerNorm(transformer_depth),
            nn.ReLU()
        )

        # Use sinusoidal positional encoding
        self.pos_encoder = PositionalEncoding(transformer_depth)

        encoder_layer = CustomTransformerEncoderLayer(
            d_model=transformer_depth,
            dim_feedforward=transformer_depth * 4,
            nhead=nhead,
            dropout=transformer_dropout,
            activation='gelu',
            batch_first=True,
            norm_first=True     # LayerNorm first
        )
        self.transformer_encoder = CustomTransformerEncoder(encoder_layer,
num_layers=num_encoders)
        self.attention_pool = MultiHeadPool(transformer_depth, nhead, transform-
er_dropout)
```

```
    def forward(self, x):
        # CNN Feature extraction
        x = self.cnn(x)  # (batch, cnn_units, seq)
        x = x.permute(0, 2, 1)  # (batch, seq, cnn_units)

        # RNN processing
        q, _ = self.rnn(x)  # (batch, seq, rnn_units*2)

        # Project to transformer dimensions
        x = self.x_proj(x)  # (batch, seq, d_model)
        q = self.q_proj(q)  # (batch, seq, d_model)

        # Add positional encoding
        x = self.pos_encoder(x)
        q = self.pos_encoder(q)

        # Transformer processing
        x = self.transformer_encoder(x, q)  # (batch, seq, d_model)

        # Multi-Head Attention pooling
        x = self.attention_pool(x)  # (batch, d_model)

        return x

    def __str__(self):
        model_describe = ""
        model_describe += "CNN: " + str(self.cnn) + "\n" * 2
        model_describe += "RNN: " + str(self.rnn) + "\n" * 2
        model_describe += "Projection X to transformer depth" + str(self.x_proj) +
"\n" * 2
        model_describe += "Projection Q to transformer depth" + str(self.q_proj) +
"\n" * 2
        model_describe += str(self.pos_encoder) + "\n" * 2
        model_describe += "Transformer Encoder: " + str(self.transformer_encoder) +
"\n"
        model_describe += "Attention pooling: " + str(self.attention_pool) + "\n" * 2
        return model_describe
```

# ПРИЛОЖЕНИЕ Д – Листинг программы загрузки данных

```
import torch
import torchaudio
import json
import numpy as np
import pandas as pd
from collections.abc import Iterator
from abc import abstractmethod

from random import randint
from ast import literal_eval
import os
from datetime import datetime

import torchaudio.transforms as T
from torch_audiomentations import Compose, PitchShift, Shift
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import Dataset, DataLoader

from sklearn.base import TransformerMixin
from sklearn.model_selection import KFold, train_test_split
from sklearn.preprocessing import OneHotEncoder, MultiLabelBinarizer, LabelEncoder

from transformers import WhisperTokenizer           # Tokenizer for text sentiment
classification
from datasets import load_dataset, load_from_disk, DatasetDict

from config import *


def one_hot_encode_labels(labels: pd.DataFrame) -> pd.DataFrame:
    """
    One-hot encode the labels.

    This encoder used for 2/4/8-moods dataset.
    """
    encoder = OneHotEncoder(sparse_output=False)
    one_hot_labels = encoder.fit_transform(labels)
    classes = encoder.categories_[0].tolist()
    return one_hot_labels, classes

def multi_label_binarize(labels: pd.Series) -> pd.Series:
    """
    Multi Label Binarization encode the labels.

    This encoder used for all-moods dataset.
    """
    encoder = MultiLabelBinarizer()
    binarized_labels = encoder.fit_transform(labels)
    classes = encoder.classes_.tolist()
    return binarized_labels, classes

def label_encode(labels: pd.DataFrame) -> pd.DataFrame:
    """
    Simple label encoding.
    :return: encoded labels and initial classes list (for parsing to string back)
```

```
    """
    encoder = LabelEncoder()
    encoded_labels = encoder.fit_transform(labels)
    classes = encoder.classes_.tolist()
    return encoded_labels, classes

def melspecs_classify_collate_fn(batch):
    """
    Custom collaction function with padding to maximum sequence length inside the
batch
    """
    xs, ys = zip(*batch)
    xs = [x.permute(1, 0) for x in xs]

    # Padding by maximum seq_len.
    xs_padded = pad_sequence(xs, batch_first=True, padding_value=0.)

    # Return initial dimensions order (batch, channel, time)
    xs_padded = xs_padded.permute(0, 2, 1)
    return xs_padded, torch.tensor(ys)

def melspecs_autoencode_collate_fn(batch):
    """
    Custom collaction function with y = spec output
    """
    xs, _ = zip(*batch)
    xs = [x.permute(1, 0) for x in xs]

    # Padding by maximum seq_len.
    xs_padded = pad_sequence(xs, batch_first=True, padding_value=0.)

    # Return initial dimensions order (batch, channel, time)
    xs_padded = xs_padded.permute(0, 2, 1)

    return xs_padded, xs_padded

def plutchik_to_3class(labels: list[int]) -> int:
    """
    Parses platchick classes: [anger, anticipation, disgust, fear, joy, sadness, sur-
prise, trust])
    To the:
      0 – sad (if sadness=1 and joy=0 or sadness=0, joy=0, anger/disgust/fear=1)
      1 – happy (if joy=1 and sadness=0 or sadness=0, joy=0, surprise=1)
    """
    classes = {"joy": 4, "sadness": 5}

    happy = 1 if classes["joy"] in labels else 0
    sad = 1 if classes["sadness"] in labels else 0
    if happy == 1 and sad == 0:
        return 1  # happy
    elif sad == 1 and happy == 0:
        return 0  # sad

    return np.nan

class PadAugmentCollate:
    def __init__(self, pad_value: float, augmentation: TransformerMixin | None =
None, for_train: bool = False):
```

```
        self.pad_value = pad_value
        self.augmentation = augmentation
        self.for_train = for_train

    def __call__(self, batch):
        # batch: list of (waveform: Tensor[T], label: Tensor)
        xs, ys = zip(*batch)
        xs_padded = pad_sequence(xs, batch_first=True, pad-
ding_value=self.pad_value)  # (B, T_max)
        xs_padded = xs_padded.unsqueeze(1)  # add channel dim (B, 1, T_max)

        # Augmentation if required
        if self.for_train and (self.augmentation is not None):
            xs_padded = self.augmentation.transform(xs_padded)

        # for i in range(xs_padded.shape[0]):
        #     torchaudio.save(
        #         uri=f"F:/dataset/music/augmented{i}_{ys[i].argmax()}.wav",
        #         src=xs_padded[i],
        #         sample_rate=16000,
        #     )
        return xs_padded, torch.tensor(ys)

class HeterogeneousDataCollate:
    def __init__(self, audio_pad_value: float, audio_augmentation: TransformerMixin |
None = None, for_train: bool = False):
        self.audio_pad_value = audio_pad_value
        self.audio_augmentation = audio_augmentation
        self.for_train = for_train

    def __call__(self, batch):
        # batch: list of (waveform: Tensor[T], label: Tensor)
        xs, ys = zip(*batch)
        audio, spec = zip(*xs)
        audio_padded = pad_sequence(audio, batch_first=True, pad-
ding_value=self.audio_pad_value)  # (B, T_max)
        audio_padded = audio_padded.unsqueeze(1)  # add channel dim (B, 1, T_max)

        # Augmentation if required
        if self.for_train and (self.audio_augmentation is not None):
            xs_padded = self.audio_augmentation.transform(xs_padded)

        spec = [s.permute(1, 0) for s in spec]

        # Padding by maximum seq_len.
        spec_padded = pad_sequence(spec, batch_first=True, padding_value=0.)

        # Return initial dimensions order (batch, channel, time)
        spec_padded = spec_padded.permute(0, 2, 1)

        return (audio_padded, spec_padded), torch.tensor(ys)

class KFoldDataLoader(Iterator):
    """
    Interface of DataLoader objects for train/val folds and testing supsets. Iterable
by folds.
    """
```

```
    def __init__(self, dataset_path: str, dataset_name: str, splits: int, tar-
get_mode: str, test_size=0.2,
                 batch_size=32, num_workers=8, outputs_path='./outputs', moods="all",
random_state=None):
        """
        :param dataset_path: Path to the dataset directory.
        :param dataset_name: Name of the dataset file (depends on moods mode).
        :param splits: Number of folds.
        :param test_size: Proportion of the dataset to include in the test split.
        :param batch_size: Batch size for the DataLoader.
        :param num_workers: Number of subprocesses to use for data loading.
        :param outputs_path: Path to save tags and labels conformity.
        :param random_state: Random seed for reproducibility.
        """
        self.dataset_path = dataset_path
        self.dataset_name = dataset_name
        self.full_dataset_path = os.path.join(dataset_path, dataset_name)
        self.random_state = random_state
        self.outputs_path = outputs_path
        self.num_workers = num_workers
        self.target_mode = target_mode
        self.batch_size = batch_size
        self.test_size = test_size
        self.splits = splits
        self.classes = None
        self.moods = moods

        if num_workers == 0:
            self.prefetch_factor = None
            self.persistent_workers = False
        else:
            self.prefetch_factor = 2
            self.persistent_workers = True

        self.x_train, self.x_test, self.y_train, self.y_test =
self._load_train_test()
        kf = KFold(n_splits=splits, shuffle=True, random_state=random_state)

        self.kf_iter = kf.split(self.x_train, self.y_train)
        self.current_fold = 0
        self.start_fold = 0

    def __iter__(self):
        return self

    def _load_train_test(self) -> tuple:
        """
        Service function for loading and spliting .tsv dataset
        """
        df = pd.read_csv(self.full_dataset_path, sep='\t')

        # Select the target transformation based on the target mode.
        if self.target_mode == MULTILABEL_TARGET:
            y, classes = multi_label_binarize(df['tags'].apply(literal_eval))  # pro-
cess the labels, apply literal_eval to convert strings to lists
        # elif self.target_mode == ONE_HOT_TARGET:
        #     y, classes = one_hot_encode_labels(df['tags'].to_frame())  # process
the labels to one-hot vectors
```

```
        else:
            y, classes = label_encode(df['tags'])  # simple label encoding

        # Use dataframe next.
        y = pd.DataFrame(y)
        df = df.drop(columns=['tags'])  # remove tags column, it is not needed any-
more

        # Save classes and encoded labels conformity to json file.
        if self.random_state is not None:
            version = self.random_state
        else:
            version = datetime.now().strftime(DATE_FORMAT)

        self.classes = classes
        classes_filename = f"classes_{self.target_mode}_{self.moods}_{version}.json"
        with open(os.path.join(self.outputs_path, classes_filename), "w", encod-
ing="utf-8") as file:
            json.dump(classes, file, indent=4)

        x_train, x_test, y_train, y_test = train_test_split(df, y,
test_size=self.test_size, random_state=self.random_state)

        return x_train, x_test, y_train, y_test

    def __next__(self):
        while self.current_fold < self.start_fold:
            next(self.kf_iter)  # skip
            self.current_fold += 1

        self.current_fold += 1
        # Get next fold.
        train_indices, val_indices = next(self.kf_iter)

        # Get slices of X, y.
        x_train = self.x_train.iloc[train_indices]
        y_train = self.y_train.iloc[train_indices]

        x_val = self.x_train.iloc[val_indices]
        y_val = self.y_train.iloc[val_indices]

        # Build loaders and return them.
        train_loader = self._get_loader(x_train, y_train, for_train=True)
        val_loader = self._get_loader(x_val, y_val, for_train=False)

        return train_loader, val_loader

    def get_train_len(self):
        return len(self.x_train)

    def set_start(self, start):
        """
        Required for loading from checkpoints.
        """
        self.start_fold = start

    @abstractmethod
    def _get_loader(self, x, y, for_train=True):
```

```
        """Get loader of batched data"""

    def get_test_loader(self):
        return self._get_loader(self.x_test, self.y_test, for_train=False),
self.classes


    def __len__(self):
        return self.splits


class KFoldSpecsDataLoader(KFoldDataLoader):
    """
    Load the dataset and create DataLoader objects for train/val folds and testing
supsets.

    Splits train/test as 1 - test_size / test_size (0.8 / 0.2 by default).

    Applyes transformation transform_specs for spectrogramms.
    Truncates/paddes spectrogrammss if required (if min_seq_len or max_seq_len provid-
ed).
    """
    def __init__(self, dataset_path: str, dataset_name: str, splits: int, tar-
get_mode: str, min_seq_len: None, max_seq_len: None,
                 pad_value=-90., test_size=0.2, batch_size=32, transform_specs: Trans-
formerMixin = None, use_augmentation = True,
                 num_workers=8, outputs_path='./outputs', moods="all", ran-
dom_state=None):
        """
        :param max_seq_len: Constant max sequence length. Spectrograms with another
length will be truncated to this length. Optional
        :param min_seq_len: Constant min sequence length. Spectrograms with another
length will be padded to this length. Optional
        :param pad_value: Pad value for cases where padding of spectrogram required.
        :param transform_specs: sklearn transformer for preprocessing spectrograms.
Optional.
        """
        super().__init__(dataset_path, dataset_name, splits, target_mode, test_size,
batch_size, num_workers, outputs_path, moods, random_state)

        self.transform_specs = transform_specs
        self.min_seq_len = min_seq_len
        self.max_seq_len = max_seq_len
        self.pad_value = pad_value

        self.augmentation = None
        if use_augmentation:
            self.augmentation = SpecAugment(mask_value=pad_value)

    def __iter__(self):
        return self

    def __next__(self):
        return super().__next__()

    def _get_loader(self, x, y, for_train=True):
        if self.target_mode == AUTOENCODER_TARGET:
            collate_fn = melspecs_autoencode_collate_fn
```

```
        else:
            collate_fn = melspecs_classify_collate_fn

        dataset = MelspecsDataset(x, y, dataset_path=self.dataset_path, trans-
form_specs=self.transform_specs,
                                  augmentation=self.augmentation,
max_seq_len=self.max_seq_len, training=for_train,
                                  min_seq_len=self.min_seq_len,
pad_value=self.pad_value)

        return DataLoader(
            dataset,
            batch_size=self.batch_size,
            shuffle=True,
            pin_memory=True,
            num_workers=self.num_workers,
            prefetch_factor=self.prefetch_factor,
            persistent_workers=self.persistent_workers,
            collate_fn=collate_fn,
        )

class KFoldFeaturesDataLoader(KFoldDataLoader):
    """
    Load the audio features dataset and create DataLoader objects for train/val folds
and testing supsets.

    Splits train/test as 1 - test_size / test_size (0.8 / 0.2 by default).
    """
    def __init__(self, dataset_path: str, dataset_name: str, splits: int, tar-
get_mode: str, test_size=0.2, batch_size=32,
                 num_workers=8, outputs_path='./outputs', moods="all", ran-
dom_state=None):
        super().__init__(dataset_path, dataset_name, splits, target_mode, test_size,
batch_size, num_workers, outputs_path, moods, random_state)

    def __iter__(self):
        return self

    def __next__(self):
        return super().__next__()

    def _get_loader(self, x, y, for_train=True):
        dataset = FeaturesDataset(x, y)

        return DataLoader(
            dataset,
            batch_size=self.batch_size,
            shuffle=True,
            pin_memory=True,
            num_workers=self.num_workers,
            prefetch_factor=self.prefetch_factor,
            persistent_workers=self.persistent_workers,
        )

    def get_input_features_number(self):
        """
        Returns number of input features.
        """
```

```
        return self.x_train.shape[1] - 2  # -2 because of track_id and path columns


class FeaturesDataset(Dataset):
    """
    Loader of audio features dataset.
    """
    def __init__(self, x: pd.DataFrame, y: pd.DataFrame):
        self.x = x.drop(columns=['track_id', 'path'])
        self.y = y

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        inputs = torch.tensor(self.x.iloc[idx].to_numpy(), dtype=torch.float)  # size
(num_features,)
        label = torch.tensor(self.y.iloc[idx].to_numpy(), dtype=torch.float)   # size
(num_classes,)
        return inputs, label


class KFoldRawAudioDataLoader(KFoldDataLoader):
    """
    Load the dataset and create DataLoader objects for train/val folds and testing
supsets.

    Splits train/test as 1 - test_size / test_size (0.8 / 0.2 by default).

    Applyes transformation transform_specs for raw audiodata.
    Truncates/paddes audio if required (if min_seq_len or max_seq_len provided).
    """
    def __init__(self, dataset_path: str, dataset_name: str, splits: int, tar-
get_mode: str, min_seq_len: None, max_seq_len: None,
                 pad_value=0., test_size=0.2, batch_size=32, transform_audio: Trans-
formerMixin = None, use_augmentation = True,
                 sample_rate=22050, num_workers=8, outputs_path='./outputs',
moods="all", random_state=None):
        """
        :param max_seq_len: Constant max sequence length. Spectrograms with another
length will be truncated to this length. Optional
        :param min_seq_len: Constant min sequence length. Spectrograms with another
length will be padded to this length. Optional
        :param pad_value: Pad value for cases where padding of spectrogram required.
        :param transform_specs: sklearn transformer for preprocessing spectrograms.
Optional.
        """
        super().__init__(dataset_path, dataset_name, splits, target_mode, test_size,
batch_size, num_workers, outputs_path, moods, random_state)
        self.transform_audio = transform_audio
        self.min_seq_len = min_seq_len
        self.max_seq_len = max_seq_len
        self.sample_rate = sample_rate
        self.pad_value = pad_value

        self.augmentation = None
        if use_augmentation:
            self.augmentation = AudioAugment(sample_rate=sample_rate)
```

```
    def __iter__(self):
        return self

    def __next__(self):
        return super().__next__()

    def _get_loader(self, x, y, for_train=True):
        collate_fn = PadAugmentCollate(pad_value=self.pad_value, augmenta-
tion=self.augmentation, for_train=for_train)

        dataset = RawAudioDataset(x, y, dataset_path=self.dataset_path, sam-
ple_rate=self.sample_rate,
                                  max_seq_len=self.max_seq_len,
min_seq_len=self.min_seq_len,
                                  pad_value=self.pad_value)

        return DataLoader(
            dataset,
            batch_size=self.batch_size,
            shuffle=True,
            pin_memory=True,
            num_workers=self.num_workers,
            prefetch_factor=self.prefetch_factor,
            persistent_workers=self.persistent_workers,
            collate_fn=collate_fn,
        )

class KFoldHeterogeneousDataLoader(KFoldDataLoader):
    """
    Load the dataset and create DataLoader objects for train/val folds and testing
supsets.
    Splits train/test as 1 - test_size / test_size (0.8 / 0.2 by default).
    """
    def __init__(self,
                 dataset_path: str,
                 dataset_name: str,
                 splits: int,
                 target_mode: str,
                 min_audio_seq_len: None,
                 max_audio_seq_len: None,
                 min_spec_seq_len: None,
                 max_spec_seq_len: None,
                 audio_pad_value=0.,
                 spec_pad_value=-90.,
                 test_size=0.2,
                 batch_size=32,
                 transform_audio: TransformerMixin = None,
                 transform_spec: TransformerMixin = None,
                 use_augmentation = True,
                 sample_rate=22050,
                 num_workers=8,
                 outputs_path='./outputs',
                 moods="all",
                 random_state=None
        ):
        super().__init__(dataset_path, dataset_name, splits, target_mode, test_size,
batch_size, num_workers, outputs_path, moods, random_state)
```

```
        self.transform_audio = transform_audio
        self.transform_spec = transform_spec
        self.min_audio_seq_len = min_audio_seq_len
        self.max_audio_seq_len = max_audio_seq_len
        self.min_spec_seq_len = min_spec_seq_len
        self.max_spec_seq_len = max_spec_seq_len
        self.sample_rate = sample_rate
        self.audio_pad_value = audio_pad_value
        self.spec_pad_value = spec_pad_value

        self.augmentation_audio = None
        self.augmentation_spec = None
        if use_augmentation:
            self.augmentation_audio = AudioAugment(sample_rate=sample_rate)
            self.augmentation_spec = SpecAugment()

    def __iter__(self):
        return self

    def __next__(self):
        return super().__next__()

    def _get_loader(self, x, y, for_train=True):
        collate_fn = HeterogeneousDataCollate(audio_pad_value=self.audio_pad_value,
audio_augmentation=self.augmentation_audio, for_train=for_train)

        dataset = HeterogeneousDataset(x, y, self.dataset_path,
                                       self.min_audio_seq_len,
                                       self.max_audio_seq_len,
                                       self.audio_pad_value,
                                       self.min_spec_seq_len,
                                       self.min_spec_seq_len,
                                       self.spec_pad_value,
                                       training=for_train,
                                       sample_rate=self.sample_rate,
                                       transform_specs=self.transform_spec,
                                       specs_augmentation=self.augmentation_spec)

        return DataLoader(
            dataset,
            batch_size=self.batch_size,
            shuffle=True,
            pin_memory=True,
            num_workers=self.num_workers,
            prefetch_factor=self.prefetch_factor,
            persistent_workers=self.persistent_workers,
            collate_fn=collate_fn,
        )


class TextSentimentDataLoader:
    """
    Load the dataset of text sentiments and create DataLoader objects for train/val
folds and testing supsets.
    Splits train/test as 1 - test_size / test_size (0.8 / 0.2 by default).

    Uses Hugging Face Datasets library to load the multilanguage dataset. Tokenize
text with WhisperTokenizer:
```

```
        (for models TextExtractor -> LirycsSentimentTransformer consistency).
    """
    def __init__(self, source_dataset_path: str, dataset_cashed_path: str,
batch_size=32,
                 num_classes=3, num_workers=8, max_length: int = 128,
                 whisper_model_name: str = "openai/whisper-small", ran-
dom_state=None):
        self.tokenizer = WhisperTokenizer.from_pretrained(whisper_model_name)
        self.source_dataset_path = source_dataset_path
        self.max_length = max_length
        self.batch_size = batch_size
        self.num_workers = num_workers
        self.random_state = random_state
        self.num_classes = num_classes

        if num_workers == 0:
            self.prefetch_factor = None
            self.persistent_workers = False
        else:
            self.prefetch_factor = 2
            self.persistent_workers = True

        print("Tokenizer vocab size:", self.tokenizer.vocab_size)

        if os.path.exists(dataset_cashed_path):
            print("Load cashed dataset...")
            self.dataset = load_from_disk(dataset_cashed_path)
        else:
            print("Loading, tokenizing and cashing dataset...")
            xed_df = self.load_xed_dataset()
            xed_df["sentiment_label"] = xed_df["labels"].apply(plutchik_to_3class)
            xed_df = xed_df.dropna()
            xed_df = xed_df[["sentence", "sentiment_label", "language"]]

            train_val_df, test_df = train_test_split(
                xed_df,
                test_size=0.2,
                stratify=xed_df["sentiment_label"],
                random_state=self.random_state
            )
            train_df, val_df = train_test_split(
                train_val_df,
                test_size=0.25,
                stratify=train_val_df["sentiment_label"],
                random_state=self.random_state
            )

            print("Train size:", len(train_df))
            print("Valid size:", len(val_df))
            print("Test size:", len(test_df))

            train_ds = Dataset.from_pandas(train_df)
            val_ds   = Dataset.from_pandas(val_df)
            test_ds  = Dataset.from_pandas(test_df)

            # Удалим лишний индекс столбца (HF может создать 'index' при from_pandas)
            train_ds = train_ds.remove_columns(["__index_level_0__"])
            val_ds   = val_ds.remove_columns(["__index_level_0__"])
```

```
            test_ds  = test_ds.remove_columns(["__index_level_0__"])

            self.dataset = DatasetDict({
                "train": train_ds,
                "validation": val_ds,
                "test": test_ds
            })

            self.dataset = self.dataset.map(self.preprocess, re-
move_columns=self.dataset["train"].column_names)
            self.dataset.save_to_disk(dataset_cashed_path)

        from collections import Counter
        lang_counts = Counter(self.dataset["train"]["language"])
        print("Available languages in training set:", lang_counts)
        self.dataset.set_format(type="torch", columns=["input_ids", "attention_mask",
"labels", "language"])
        lang_counts = Counter(self.dataset["train"]["language"])
        print("Available languages in training set:", lang_counts)

    def load_xed_dataset(self):
        data_frames = []

        for lang, fname in [("en", "en-annotated.tsv"), ("fi", "fi-annotated.tsv")]:
            path_annot = os.path.join(self.source_dataset_path, "AnnotatedData",
fname)
            df = pd.read_csv(path_annot, sep="\t", header=None, names=["sentence",
"labels"])
            df["language"] = lang
            df["labels"] = df["labels"].apply(lambda s: [int(x) for x in s.split(',
')])
            data_frames.append(df)

        aligned_dir = os.path.join(self.source_dataset_path, "Projections")
        for fname in os.listdir(aligned_dir):
            if not fname.endswith(".tsv"):
                continue
            lang = fname.replace("-projections.tsv", "")
            path_lang = os.path.join(aligned_dir, fname)
            df = pd.read_csv(path_lang, sep="\t", header=None, names=["sentence",
"labels"])
            df["language"] = lang
            df["labels"] = df["labels"].apply(lambda s: [int(x) for x in s.split(',
')])
            data_frames.append(df)

        full_df = pd.concat(data_frames, ignore_index=True)
        return full_df

    def preprocess(self, example):
        text = " " + example["sentence"].strip()
        enc = self.tokenizer(
            text,
            padding="max_length",
            truncation=True,
            max_length=self.max_length,
            return_tensors="pt",
        )
```

```
        return {
            "input_ids": enc["input_ids"].squeeze(0),               # tokenized indi-
ces from whisper dictionary
            "attention_mask": enc["attention_mask"].squeeze(0),     # attention mask
for padding masking
            "labels": example["sentiment_label"],                   # target labels
            "language": example["language"],                        # language of the
text
        }

    def get_dataloader(self, dataset_split: str):
        loader = DataLoader(
            self.dataset[dataset_split],
            batch_size=self.batch_size,
            shuffle=True,
            pin_memory=True,
            num_workers=self.num_workers,
            prefetch_factor=self.prefetch_factor,
            persistent_workers=self.persistent_workers,
        )
        return loader

    def get_dataloader_for_language(self, dataset_split: str, languages: list):
        filtered = self.dataset[dataset_split].filter(lambda ex: ex["language"] in
languages)

        loader = DataLoader(
            filtered,
            batch_size=self.batch_size,
            shuffle=True,
            pin_memory=True,
            num_workers=self.num_workers,
            prefetch_factor=self.prefetch_factor,
            persistent_workers=self.persistent_workers,
        )
        return loader


class RawAudioDataset(Dataset):
    """
    Loader of raw audio dataset.
    """
    def __init__(self, x, y, dataset_path, min_seq_len, max_seq_len, pad_value, sam-
ple_rate = 22050):
        self.x = x
        self.y = y
        self.dataset_path = dataset_path
        self.min_seq_len = min_seq_len
        self.max_seq_len = max_seq_len
        self.pad_value = pad_value
        self.sample_rate = sample_rate
        self.resampler = None

        self.start_skip_frames = sample_rate * 1  # skip a first second in audio
        self.end_skip_frames = sample_rate * 1  # skip a last second in audio

    def __len__(self):
        return len(self.x)
```

```
    def __getitem__(self, idx):
        row = self.x.iloc[idx]
        audio_path = os.path.join(self.dataset_path, row['path'])

        # Number of input samples.
        segment_len = self.max_seq_len or self.sample_rate  # if not specified -> 1
second

        # Get total len of audio.
        info = torchaudio.info(audio_path)
        total_len = info.num_frames

        # Select random part of audio.
        if total_len > segment_len + self.end_skip_frames:
            start = torch.randint(self.start_skip_frames,
                total_len - segment_len - self.end_skip_frames + 1, (1,)
            ).item()
        else:
            start = self.start_skip_frames

        audio, sr = torchaudio.load(
            audio_path,
            frame_offset=start,
            num_frames=segment_len,
            normalize=True
        )

        # Resample if needed (often downsample)
        if sr != self.sample_rate:
            if self.resampler is None or self.resampler.orig_freq != sr:
                self.resampler = T.Resample(orig_freq=sr, new_freq=self.sample_rate)
            audio = self.resampler(audio)

        # Convert to mono if stereo
        if audio.ndim == 2:
            audio = audio.mean(dim=0)

        label = torch.tensor(self.y.iloc[idx], dtype=torch.float)  # size
(num_classes,)
        return audio, label


class MelspecsDataset(Dataset):
    """
    Loader of mel-spectrograms dataset.
    """
    def __init__(self, x, y, dataset_path, min_seq_len, max_seq_len, pad_value,
training = True,
                 transform_specs: TransformerMixin = None, augmentation: Transform-
erMixin = None):
        self.x = x
        self.y = y
        self.dataset_path = dataset_path
        self.training = training
        self.augmentation = augmentation
        self.transform_specs = transform_specs
        self.min_seq_len = min_seq_len
```

```
        self.max_seq_len = max_seq_len
        self.pad_value = pad_value

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        row = self.x.iloc[idx]

        spec = np.load(os.path.join(self.dataset_path, row['melspecs_path']))

        if self.min_seq_len is not None and spec.shape[1] < self.min_seq_len:
            spec = self.pad_spec(spec)  # pad spec if required
        elif self.max_seq_len is not None and spec.shape[1] > self.max_seq_len:
            spec = self.truncate_spec(spec)  # truncate spec if required

        # Apply augmentation. Use audio augmentation before scaling/normalization!
        if self.augmentation and self.training:
            spec = self.augmentation.transform(spec)

        # Apply transformation to the mel spectrogram if specified
        if self.transform_specs:
            spec = self.transform_specs.transform(spec)

        spec = torch.from_numpy(spec).float()  # size (num_mels, num_frames)
        label = torch.tensor(self.y.iloc[idx], dtype=torch.float)  # size
(num_classes,)
        return spec, label

    def pad_spec(self, spec: np.ndarray) -> np.ndarray:
        """
        Pads spectrogramms to min_seq_len length. Length is second dimension.
        """
        pad_width = ((0, 0), (0, self.min_seq_len - spec.shape[1]))
        return np.pad(spec, pad_width=pad_width, mode='constant', con-
stant_values=self.pad_value)

    def truncate_spec(self, spec: np.ndarray) -> np.ndarray:
        """
        Truncates spectrogramms to max_seq_len length. Length is second dimension.

        Makes random correct choise of start point.
        """
        start = randint(0, spec.shape[1] - self.max_seq_len)
        return spec[:, start:start + self.max_seq_len]


class HeterogeneousDataset(Dataset):
    def __init__(self, x, y, dataset_path, min_audio_seq_len, max_audio_seq_len, au-
dio_pad_value,
                 min_spec_seq_len, max_spec_seq_len, spec_pad_value, training=True,
sample_rate = 22050,
                 transform_specs: TransformerMixin = None, specs_augmentation: Trans-
formerMixin = None):
        self.x = x
        self.y = y
        self.dataset_path = dataset_path
        self.min_audio_seq_len = min_audio_seq_len
```

```
        self.max_audio_seq_len = max_audio_seq_len
        self.min_spec_seq_len = min_spec_seq_len
        self.max_spec_seq_len = max_spec_seq_len
        self.audio_pad_value = audio_pad_value
        self.spec_pad_value = spec_pad_value
        self.sample_rate = sample_rate
        self.training = training
        self.resampler = None

        self.transform_specs = transform_specs
        self.specs_augmentation = specs_augmentation

        self.start_skip_frames = sample_rate * 1  # skip a first second in audio
        self.end_skip_frames = sample_rate * 1  # skip a last second in audio

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        # AUDIO
        row = self.x.iloc[idx]
        audio_path = os.path.join(self.dataset_path, row['path'])

        # Number of input samples.
        segment_len = self.max_audio_seq_len or self.sample_rate  # if not specified
-> 1 second

        # Get total len of audio.
        info = torchaudio.info(audio_path)
        total_len = info.num_frames

        # Select random part of audio.
        if total_len > segment_len + self.end_skip_frames:
            start = torch.randint(self.start_skip_frames,
                total_len - segment_len - self.end_skip_frames + 1, (1,)
            ).item()
        else:
            start = self.start_skip_frames

        audio, sr = torchaudio.load(
            audio_path,
            frame_offset=start,
            num_frames=segment_len,
            normalize=True
        )

        # Resample if needed (often downsample)
        if sr != self.sample_rate:
            if self.resampler is None or self.resampler.orig_freq != sr:
                self.resampler = T.Resample(orig_freq=sr, new_freq=self.sample_rate)
            audio = self.resampler(audio)

        # Convert to mono if stereo
        if audio.ndim == 2:
            audio = audio.mean(dim=0)

        # MELSPEC
        spec = np.load(os.path.join(self.dataset_path, row['melspecs_path']))
```

```
        if self.min_spec_seq_len is not None and spec.shape[1] <
self.min_spec_seq_len:
            spec = self.pad_spec(spec)  # pad spec if required
        elif self.max_spec_seq_len is not None and spec.shape[1] >
self.max_spec_seq_len:
            spec = self.truncate_spec(spec)  # truncate spec if required

        # Apply augmentation. Use audio augmentation before scaling/normalization!
        if self.specs_augmentation and self.training:
            spec = self.specs_augmentation.transform(spec)

        # Apply transformation to the mel spectrogram if specified
        if self.transform_specs:
            spec = self.transform_specs.transform(spec)

        spec = torch.from_numpy(spec).float()  # size (num_mels, num_frames)
        label = torch.tensor(self.y.iloc[idx], dtype=torch.float)  # size
(num_classes,)
        return ((audio, spec), label)

    def pad_spec(self, spec: np.ndarray) -> np.ndarray:
        """
        Pads spectrogramms to min_seq_len length. Length is second dimension.
        """
        pad_width = ((0, 0), (0, self.min_spec_seq_len - spec.shape[1]))
        return np.pad(spec, pad_width=pad_width, mode='constant', con-
stant_values=self.spec_pad_value)

    def truncate_spec(self, spec: np.ndarray) -> np.ndarray:
        """
        Truncates spectrogramms to max_seq_len length. Length is second dimension.

        Makes random correct choise of start point.
        """
        start = randint(0, spec.shape[1] - self.max_spec_seq_len)
        return spec[:, start:start + self.max_spec_seq_len]


class SpecAugment(TransformerMixin):
    """
    Spectrograms augmentation. Performs random Gaussian noise, applies masking and
gain with some probability.
    Works with unscaled spectrograms in decibels (dB).
    """
    def __init__(self, freq_mask_param=6, time_mask_param=64, mask_value=-90.,
p_f=0.3, p_t=0.3, p_g=0.3):
        self.p_f = p_f
        self.p_t = p_t
        self.p_g = p_g

        self.mask_value = mask_value
        self.freq_mask = T.FrequencyMasking(freq_mask_param)
        self.time_mask = T.TimeMasking(time_mask_param)
        self.randomGain = RandomGain(min_gain=-5, max_gain=5)   # gain in db
        self.gaussianNoise = GaussianNoise(std=0.4)             # noise in db

    def transform(self, spec: np.ndarray) -> np.ndarray:
```

```
        spec = self.gaussianNoise(spec)

        # Apply gain with probability p_g.
        if np.random.rand() < self.p_g:
            spec = self.randomGain(spec)

        spec_tensor = torch.from_numpy(spec).float()

        # Apply frequency masking with probability p_f.
        if np.random.rand() < self.p_f:
            spec_tensor = self.freq_mask(spec_tensor, mask_value=self.mask_value)

        # Apply time masking with probability p_t.
        if np.random.rand() < self.p_t:
            spec_tensor = self.time_mask(spec_tensor, mask_value=self.mask_value)

        return spec_tensor.numpy()


class AudioAugment(TransformerMixin):
    def __init__(self, sample_rate, p_f=0.3, p_g=0.3, p_e=0.3):
        """
        Batch-level augmentation (expects batch x channels x time)
        """
        self.sample_rate = sample_rate
        self.p_f = p_f
        self.p_g = p_g
        self.p_e = p_e

        self.fade = T.Fade(fade_in_len=int(0.1 * sample_rate), fade_out_len=int(0.1 *
sample_rate))  # 10% fade in/out

    def transform(self, waveform: torch.Tensor) -> torch.Tensor:
        # Random gain
        if torch.rand(1).item() < self.p_g:
            # Random gain factor between 0.6 and 1.1
            gain_factor = torch.empty(1).uniform_(0.6, 1.1).item()
            vol = T.Vol(gain_factor, gain_type="amplitude")
            waveform = vol(waveform)

        # Random effects
        waveform = self.apply_random_effects(waveform)

        # Fade in/out
        if torch.rand(1).item() < self.p_f:
            waveform = self.fade(waveform)

        # Gaussian noise
        waveform = self.apply_noise(waveform)

        return waveform

    def apply_random_effects(self, waveform):
        augment = Compose(
            transforms=[
                PitchShift(min_transpose_semitones=-4, max_transpose_semitones=4,
p=self.p_e, sample_rate=self.sample_rate, output_type='tensor'),
```

```
                # Shift(min_shift=-1000, max_shift=1000, p=self.p_e, sam-
ple_rate=self.sample_rate, output_type='tensor'),
            ],
            output_type='tensor'
        )

        return augment(waveform, sample_rate=self.sample_rate)

    def apply_noise(self, waveform):
        max_amp = 0.001
        random_noise_amp = torch.randn(1).item() * max_amp
        noise = torch.randn_like(waveform) * random_noise_amp
        return waveform + noise

    def apply_gain(self, waveform):
        gain = torch.empty(1).uniform_(-0.1, 0.1).item()
        return waveform + gain

class RandomGain():
    """
    Applyes random gain to the spectrogram.
    """
    def __init__(self, min_gain, max_gain):
        self.min_gain = min_gain
        self.max_gain = max_gain

    def __call__(self, spec: np.ndarray) -> np.ndarray:
        gain = np.random.uniform(self.min_gain, self.max_gain)
        return spec + gain

class GaussianNoise():
    """
    Applyes random noise to the data.
    """
    def __init__(self, mean=0.0, std=1.0):
        self.mean = mean
        self.std = std

    def __call__(self, data: np.ndarray) -> np.ndarray:
        noise = np.random.normal(self.mean, self.std, data.shape)
        return data + noise
```

# ПРИЛОЖЕНИЕ Е – Листинг серверной программы

```
import os
import re
import requests
from rest_framework import status
from dotenv import load_dotenv

load_dotenv()

def extract_track_id(url):
    pattern = r'jamendo\.com/track/(\d+)'
    match = re.search(pattern, url)
    return match.group(1) if match else None

class Prediction:
    def __init__(self):
        self.client_id = os.getenv('JAMENDO_CLIENT_ID')

    def _form_ok_response(self, response):
        result_dict = {}
        return_dict = {}

        data = response.json()

        # Form result dictionary and send response.
        probs_dict = {}
        classes = data['classes']
        probs = data['probs']

        for target_class, prob in zip(classes, probs):
            probs_dict[target_class] = prob

        result_dict['predict'] = data['predict']
        result_dict['probs'] = probs_dict

        return_dict['response'] = result_dict
        return_dict['status'] = status.HTTP_200_OK

    def _form_not_ok_response(self, text, status_code):
        return_dict = {}
        message = f"Exception while prediction: {text}"
        return_dict['response'] = text
        return_dict['status'] = status_code
        print(message)
        return return_dict

    def _form_bad_request(self, text):
        return self._form_not_ok_response(text, status.HTTP_400_BAD_REQUEST)

    def _form_internal_server_error(self, text):
        return self._form_not_ok_response(text, sta-
tus.HTTP_500_INTERNAL_SERVER_ERROR)

    def _form_forbidden(self, text):
        return self._form_not_ok_response(text, status.HTTP_403_FORBIDDEN)
```

```
    def _form_not_found(self, text):
        return self._form_not_ok_response(text, status.HTTP_404_NOT_FOUND)

    def predict_by_file(self, request):
        # In this request file required!
        if 'audio' not in request.FILES:
            return self._form_bad_request("Аудио-файл не был найден в запросе!")

        audio_file = request.FILES['audio']

        # Send audio file to the model for prediction.
        try:
            response = requests.post(
                'http://localhost:5000/api/model/predict',
                files={'file': (audio_file.name, audio_file, au-
dio_file.content_type)}
            )
        except Exception as e:
            print("Error while send prediction to model: ", e)
            return self._form_internal_server_error("Модель недоступна. Попробуйте
позже!")

        if response.status_code != 200:
            print("Model prediction failed with status code: ", response.status_code)
            return self._form_internal_server_error("Модель недоступна. Попробуйте
позже!")

        return self._form_ok_response(response)

    def predict_with_link(self, request):
        audio_url = request.data.get('link')
        if not audio_url:
            return self._form_bad_request("Ссылка на аудио не была найдена в
запросе!")

        # Get track_id from link
        track_id = extract_track_id(audio_url)

        if track_id is None or not track_id.isdigit():
            return self._form_bad_request(f"В ссылке не был найден номер трека. Ожи-
даемая ссылка: 'https://www.jamendo.com/track/<track_id>/...'; передано: {au-
dio_url}")

        if not self.client_id:
            print("Client ID not found in environment!")
            return self._form_internal_server_error("На сервере произошла временная
ошибка. Попробуйте позже!")

        # Get track info
        track_info_url =
f'https://api.jamendo.com/v3.0/tracks?client_id={self.client_id}&id={track_id}'
        try:
            response = requests.get(track_info_url)
        except Exception as e:
            print("Error while send prediction to Jamendo: ", e)
            return self._form_internal_server_error("Сервера Jamendo недоступны, по-
пробуйте позже!")
```

```
        data = response.json()

        if not data['results']:
            return self._form_not_found(f"Трек {track_id} не был найден.")

        # Check if there download available
        track = data['results'][0]
        download_url = track.get('audiodownload')
        if not track.get('audiodownload_allowed') or not download_url:
            return self._form_forbidden(f"Загрузка этого трека не разрешена автором.
Простите за неудобства.")

        # Try to load file
        try:
            audio_response = requests.get(download_url, stream=True)
        except Exception as e:
            print("Error while send prediction to Jamendo: ", e)
            return self._form_internal_server_error("Сервера Jamendo недоступны, по-
пробуйте позже!")

        if audio_response.status_code != 200:
            return self._form_internal_server_error("При загрузке файла на сервере
произошла ошибка! Попробуйте позже!")

        # Send audio file to the model.
        try:
            response = requests.post(
                'http://localhost:5000/api/model/predict',
                files={'file': ('audio.mp3', audio_response.raw, 'audio/mpeg')}
            )
        except Exception as e:
            print("Error while send prediction to model: ", e)
            return self._form_internal_server_error("Модель недоступна. Попробуйте
позже!")

        if response.status_code != 200:
            print("Model prediction failed with status code: ", response.status_code)
            return self._form_internal_server_error("Модель недоступна. Попробуйте
позже!")

        return self._form_ok_response(response)
```

# ПРИЛОЖЕНИЕ Ж – Листинг клиентского приложения

```
import Home from './components/home/Home';
import classes from './app.module.css'

function App() {
    return (
        <div className={classes.app}>
            <header>
                <h1 className={classes.header}>
                    Определение эмоциональной окраски музыкальных произведений
                </h1>
            </header>

            <main>
                <Home />
            </main>
        </div>
    )
}

export default App;

import { DarkButton } from '../ui/DarkButton'
import { FileUpload, LinkUpload } from '../ui/Inputs';
import { AlertModal, ResultModal } from '../ui/Modal';
import { DarkSelector } from '../ui/DarkSelector'
import { CardCarousel } from '../cards/CardsCarousel';
import classes from "./home.module.css";

import { useState } from "react";

export default function Home() {
    const loadCards = {
        local: "local",
        jamendo: "jamendo",
    }

    const [loaded, setLoaded] = useState('');
    const [model, setModel] = useState('1.0');
    const [currentCard, setCurrentCard] = useState(loadCards.local)
    const [localFile, setLocalFile] = useState(null);
    const [jamendoLink, setJamendoLink] = useState('');

    const [isAlertOpen, setIsAlertOpen] = useState(false);
    const [alertText, setAlertText] = useState('');

    const [isResultsOpen, setIsResultOpen] = useState(false);
    const [predict, setPredict] = useState('unknown');
    const [probs, setProbs] = useState(null);

    const onCardChange = (card) => {
        switch (card) {
            case loadCards.local:
                if (localFile == null) setLoaded(false);
                else setLoaded(true);
                break;
```

```
            case loadCards.jamendo:
                if (jamendoLink.trim()) setLoaded(true);
                else setLoaded(false);
                break;
            default:
                console.log("Invalid card! ", card);
                setLoaded(false);
                return;
        }

        setCurrentCard(card);
    };

    const onModelChange = (select) => {
        setModel(select.target.value)
    }

    const onLocalFileUpload = (file) => {
        setLoaded(true);
        setLocalFile(file);
        console.log('Загружен файл:', localFile);
    };

    const onJamendoLinkChange = (link) => {
        setJamendoLink(link)

        if (link.trim()) {
            setLoaded(true);
        } else {
            setLoaded(false);
        }
    };

    const processPredict = (result) => {
        setPredict(result["predict"]);
        setProbs(result["probs"]);

        setIsResultOpen(true);
    }

    const sendFilePredictionRequest = async (file, model) => {
        const formData = new FormData();
        formData.append("audio", file);
        formData.append("model_version", model);

        try {
            const response = await fetch("http://10.0.0.2/api/predict/", {
                method: "POST",
                body: formData,
            });

            if (!response.ok) {
                const errorText = await response.text()
                customAlert(`К сожалению, при отправке запроса произошла ошибка:
${errorText}`)
                console.error(`Ошибка: ${response.status}`);
                return null;
            }
```

```
            const result = await response.json();
            processPredict(result);
            return result;
        } catch (error) {
            console.error("Ошибка при отправке запроса:", error);
            customAlert("К сожалению, при отправке запроса произошла ошибка. Проверь-
те подключение к сети.")
            return null;
        }
    };

    const sendJamendoPredictionRequest = async (link, model) => {
        if (link.indexOf('https://www.jamendo.com/track') === -1) {
            customAlert("Ссылка должна соответствовать формату:
'https://www.jamendo.com/track/<id_track>'")
            return null
        }

        const formData = new FormData();
        formData.append("link", link);
        formData.append("model_version", model);

        try {
            const response = await fetch("http://10.0.0.2/api/predict/link/", {
                method: "POST",
                body: formData,
            });

            if (!response.ok) {
                const errorText = await response.text()
                customAlert(`К сожалению, при отправке запроса произошла ошибка:
${errorText}`)
                console.error(`Ошибка: ${response.status}`);
                return null;
            }

            const result = await response.json();
            processPredict(result);
            return result;
        } catch (error) {
            console.error("Ошибка при отправке запроса:", error);
            customAlert("К сожалению, при отправке запроса произошла ошибка. Проверь-
те подключение к сети.")
            return null;
        }
    };

    const handlePredictClick = () => {
        switch (currentCard) {
            case loadCards.local:
                if (localFile) {
                    sendFilePredictionRequest(localFile, model);
                } else {
                    console.warn("Файл не выбран!");
                    return
                }
                break;
```

```
            case loadCards.jamendo:
                if (jamendoLink !== '') {
                    sendJamendoPredictionRequest(jamendoLink, model);
                } else {
                    console.warn("Ссылка не введена!");
                    return
                }
                break;
            default:
                console.error("Неизвестная карточка:", currentCard);
                return;
        }
    };

    const customAlert = (text) => {
        setAlertText(text);
        setIsAlertOpen(true);
    };


    return (
        <div className={classes.home}>
            <p>
                Загрузите аудио-файл любым удобным способом
            </p>

            <CardCarousel onCardChange={onCardChange}>
                <div key={loadCards.local} style={{textAlign: "center", width:
"50%"}}>
                    <p>
                        Загрузка из локального хранилища
                    </p>

                    <FileUpload onFileSelect={onLocalFileUpload}/>
                </div>


                <div key={loadCards.jamendo} style={{textAlign: "center", width:
"70%"}}>
                    <p>
                        Загрузка по ссылке с
                        <a style={{marginLeft: "7px"}}
href="https://www.jamendo.com/start">
                            Jamendo
                        </a>
                    </p>

                    <LinkUpload onLinkChange={onJamendoLinkChange}/>
                </div>

            </CardCarousel>

            <div style={{display: 'flex', flexDirection: 'row'}} >
                {/* <DarkSelector
                    options={[
                        { value: 'test', label: 'specstr 19B'},
                    ]}
                    value={model}
```

```
                    onChange={onModelChange}
                    disabled={!loaded}
                /> */}

                <DarkButton onClick={handlePredictClick} disabled={!loaded}>
                    Отправить
                </DarkButton>
            </div>

            <AlertModal
                text={alertText}
                isOpen={isAlertOpen}
                onClose={() => setIsAlertOpen(false)}
            />

            <ResultModal
                predict={predict}
                probs={probs}
                isOpen={isResultsOpen}
                onClose={() => setIsResultOpen(false)}
            />
        </div>
    );
}
import React, { useState } from 'react';
import classes from './cards.module.css';

export const CardCarousel = ({ children, onCardChange }) => {
    const total = React.Children.count(children);
    const [currentIndex, setCurrentIndex] = useState(0);

    const prevCard = () => {
        const newIndex = (currentIndex - 1 + total) % total;
        onCardChange(children[newIndex].key);
        setCurrentIndex(newIndex);
    }

    const nextCard = () => {
        const newIndex = (currentIndex + 1) % total
        onCardChange(children[newIndex].key);
        setCurrentIndex(newIndex);
    }

    return (
        <div className={classes.carouselContainer}>
            <button
                className={`${classes.navButton} ${classes.prev}`}
                onClick={prevCard}
            >
                &lt;
            </button>

            <div
                className={classes.cardWrapper}
                style={{ transform: `translateX(-${currentIndex * 100}%)` }}
            >
                {React.Children.map(children, (child, index) => (
                    <div key={index} className={classes.card}>
```

```
                        {child}
                    </div>
                ))}
            </div>

            <button
                className={`${classes.navButton} ${classes.next}`}
                onClick={nextCard}
            >
                &gt;
            </button>
        </div>
    );
};

import classes from './ui.module.css';

export const DarkButton = ({ children, onClick, disabled = false }) => (
    <button onClick={onClick} className={classes.darkButton} disabled={disabled}>
        {children}
    </button>
);

import {
    Radar, RadarChart, PolarGrid, PolarAngleAxis, ResponsiveContainer
} from 'recharts';


export const EmotionRadarChart = ({ data }) => {
    const chartData = Object.entries(data).map(([emotion, value]) => ({
        emotion,
        value,
    }));

    return (
        <ResponsiveContainer width="100%" height={300} >
            <RadarChart cx="50%" cy="50%" outerRadius="65%" data={chartData}>
                <PolarGrid />
                <PolarAngleAxis
                    dataKey="emotion"
                    tick={({ payload, cx, cy }) => {
                    const angleRad = -payload.coordinate * (Math.PI / 180);
                    const radiusX = 130;
                    const radiusY = 110;

                    return (
                        <text
                            x = {cx + radiusX * Math.cos(angleRad)}
                            y = {cy + radiusY * Math.sin(angleRad)}
                            textAnchor="middle"
                            dominantBaseline="middle"
                            fill="#aaa"
                            fontSize={14}
                        >
                            {payload.value}
                        </text>
                    );
                    }}
```

```
                />
                <Radar name="Emotion" dataKey="value" stroke="#4c52ff" fill="#4c52ff"
fillOpacity={0.6} />
            </RadarChart>
        </ResponsiveContainer>
    );
};

import styles from './ui.module.css';

export const DarkSelector = ({
    options = [],
    value,
    onChange,
    disabled = false,
  }) => (
    <select
        className={styles.darkSelector}
        value={value}
        onChange={onChange}
        disabled={disabled}
    >
        {options.map(opt => (
            <option key={opt.value} value={opt.value}>
                {opt.label}
            </option>
        ))}
    </select>
);

import { useRef, useState } from 'react';
import classes from './ui.module.css';

export const FileUpload = ({ onFileSelect }) => {
    const fileInputRef = useRef(null);
    const [fileName, setFileName] = useState('');

    const handleClick = () => {
        fileInputRef.current.click();
    };

    const handleChange = (e) => {
        if (e.target.files.length > 0) {
            const file = e.target.files[0];
            setFileName(file.name);
            onFileSelect(file);
        }
    };

    return (
        <div className={classes.uploadContainer} onClick={handleClick}>
            <input
                type="file"
                accept="audio/*"
                ref={fileInputRef}
                onChange={handleChange}
                className={classes.hiddenInput}
            />
```

```
            <p>
                {fileName
                    ? `Выбран файл: ${fileName}`
                    : 'Нажмите для загрузки аудиофайла'}
            </p>
        </div>
    );
};

export const LinkUpload = ({ onLinkChange }) => {
    const [link, setLink] = useState('');

    const handleChange = (e) => {
        const newLink = e.target.value
        setLink(newLink);
        onLinkChange(newLink);
    };

    return (
        <input
            type="text"
            value={link}
            onChange={handleChange}
            placeholder="Вставьте ссылку на видео/аудио"
            className={classes.textInput}
        />
    );
};


import { useEffect } from 'react';
import classes from './ui.module.css';
import { DarkButton } from './DarkButton';
import { EmotionRadarChart } from './DarkRadar';

// Базовые цвета для эмоций в формате [R, G, B]
const EMOTION_COLORS = {
    sad:      [0, 123, 255],    // насыщенный синий
    relaxing: [102, 204, 255],  // небесно-голубой
    happy:    [0, 200, 0],      // ярко-зеленый
    energetic:[255, 165, 0],    // оранжевый
};

/**
 * Смешивает цвета на основе вероятностей эмоций
 * @param {{[key: string]: number}} data — вероятности эмоций
 * @param {{[key: string]: number[]}} colors — базовые цвета эмоций
 * @returns {string} — итоговый CSS rgb цвет
 */
function mixBackgroundColor(data, colors) {
    let r = 0, g = 0, b = 0;
    Object.entries(data).forEach(([emotion, weight]) => {
        const [cr, cg, cb] = colors[emotion] || [0, 0, 0];
        r += cr * weight;
        g += cg * weight;
        b += cb * weight;
    });
    const toByte = x => Math.min(255, Math.max(0, Math.round(x)));
```

```
    return `rgb(${toByte(r)}, ${toByte(g)}, ${toByte(b)})`;
}

export const AlertModal = ({ isOpen, onClose, text }) => {
    useEffect(() => {
        document.body.style.overflow = isOpen ? 'hidden' : 'auto';
    }, [isOpen]);

    if (!isOpen) return null;

    return (
        <div
            className={`${classes.overlay} ${isOpen ? classes.overlayOpen : ''}`}
            onClick={e => e.target === e.currentTarget && onClose()}
        >
            <div className={`${classes.modalContent} ${isOpen ? clas-
ses.modalContentOpen : ''}`}>  {/* фон не меняется здесь */}
                <div className={classes.modalBody}>
                    {text}
                </div>

                <DarkButton onClick={onClose}>
                    Закрыть
                </DarkButton>
            </div>
        </div>
    );
};

export const ResultModal = ({ isOpen, onClose, predict, probs }) => {
    useEffect(() => {
        document.body.style.overflow = isOpen ? 'hidden' : 'auto';
    }, [isOpen]);

    if (!isOpen) return null;

    const translation = {
        sad:       'грустная',
        happy:     'веселая',
        energetic: 'энергичная',
        relaxing:  'расслабляющая',
    };

    // Вычисляем цвет фона на основе вероятностей
    const moodColor = mixBackgroundColor(probs, EMOTION_COLORS);

    return (
        <div
            className={`${classes.overlay} ${isOpen ? classes.overlayOpen : ''}`}
            onClick={e => e.target === e.currentTarget && onClose()}
        >
            <div
                className={`${classes.modalContent} ${isOpen ? clas-
ses.modalContentOpen : ''}`}
                style={{ borderColor: moodColor }}
            >
                <div className={classes.modalBody}>
                    <p style={{ margin: 0, padding: 0 }}>
```

```
                        Ваша музыка 
                        <span style={{ color: moodColor, padding: 0, margin: 0 }}>
                            {translation[predict]}
                        </span>
                    </p>
                    <EmotionRadarChart data={probs} />
                </div>

                <DarkButton onClick={onClose}>
                    Назад
                </DarkButton>
            </div>
        </div>
    );
};
```