Universidad Autónoma de Madrid

Escuela Politécnica Superior

22|23

Bachelor thesis

Application of sentiment analysis on music recommendation

Javier Wang Zhou

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor as Computer Science and Engineering

BACHELOR THESIS

Application of sentiment analysis on music recommendation

Author: Javier Wang Zhou
Advisor: Alejandro Bellogín Kouki

julio 2023

Francisco Tomás y Valiente, n^o 1
Madrid, 28049
Spain

Javier Wang Zhou
Application of sentiment analysis on music recommendation

Javier Wang Zhou
C\ Francisco Tomás y Valiente Nº 11

PRINTED IN SPAIN

To my dear mother

We are not thinking machines that feel; rather, we are feeling machines that think.

Antonio Damasio

Agradecimientos

Primero que nada, agradecer a mi tutor Alejandro Bellogín por su apoyo y consejo a lo largo del año. En varias ocasiones no hubiera podido seguir con el trabajo si no fuera por su guía, y mucho menos terminarlo.

Asimismo, me gustaría expresar mi gratitud a la Escuela Politécnica Superior por ser un nexo de conocimiento, ingeniería y camaradería, donde tantos compañeros y profesores he podido conocer y de quienes aprender y depender durante la carrera.

Mi más sincero agradecimiento también a todos los amigos y compañeros que he conocido en estos años, con quienes tanto he compartido dentro y fuera de clase. En particular, a Ángel, Juan y Rodrigo, por pasar horas tras horas realizando prácticas conmigo, depurando código y aguantando mis manías.

Finalmente, gracias a mi familia, en especial a mi madre, por apoyarme este último año y a lo largo de mi vida.

Resumen

Los sistemas de recomendación musical desempeñan un papel fundamental al atender a diversas preferencias de usuarios y fomentar experiencias musicales personalizadas. De igual manera, el sentimiento puede influir profundamente en la música al dar forma a su expresión emocional y evocar estados de ánimo específicos en los oyentes. Dicho sentimiento se puede analizar mediante técnicas de procesamiento del lenguaje natural para medir emociones u opiniones expresadas en contenidos textuales, con la esperanza de aumentar la relevancia o significado al aplicarse en procesos de recomendación.

Este proyecto se adentra en el campo del análisis de sentimiento y su posible impacto en la recomendación musical, buscando mejorar los modelos de recomendación al incorporar atributos de sentimiento derivados de un conjunto de datos y un analizador de sentimientos obtenidos con este fin, con objeto de encontrar relaciones significativas entre las emociones y el gusto musical.

Como culmen de este trabajo, se desarrolla una aplicación web para mostrar las herramientas utilizadas y los modelos de mejor rendimiento. Esta plataforma ilustra el potencial de la recomendación consciente de emociones, ofreciendo una perspectiva sobre la dimensión de los sentimientos y su representación de preferencias personales.

Palabras clave

Sistemas de recomendación, recomendación de música, análisis de sentimiento, scraping de datos, incrustaciones

Abstract

Music recommendation systems play a pivotal role in catering to diverse user preferences and fostering personalized listening experiences. Likewise, sentiment can profoundly influence music by shaping its emotional expression and evoking specific moods onto listeners. This sentiment may be analyzed through natural language processing techniques to gauge emotions or opinions expressed in textual content, hopefully increasing relevance or significance when applied to the recommendation process.

This project ventures into the field of sentiment analysis and its potential impact on music recommendation, attempting to enhance recommendation models by incorporating sentiment attributes derived from a manually retrieved dataset and a bespoke sentiment analyzer — in pursuit of insightful correlations between emotion and musical taste.

As a culmination of this research endeavor, a web application is developed to showcase the applied tools and the finest-performing recommender models. This platform illustrates the potential of emotion-aware recommendation, offering a view into the dimension of sentiment and its representation of personal preferences.

Keywords

Recommender systems, music recommendation, sentiment analysis, data scraping, embeddings

Table of Contents

Lists

List of codes

List of equations

List of figures

List of tables

1

Introduction

The field of music recommendation has witnessed significant advancements in recent years, with various techniques and algorithms being employed to deliver personalized tracks to users, and applied on the most popular music platforms. However, the incorporation of sentiment analysis into music recommendation systems remains an area of research ripe for exploration. This analysis could provide additional attributes exploitable by recommender systems to perhaps improve the accuracy or true value of their recommendations.

1.1. Motivation

Music is inherently emotional and has the ability to evoke strong feelings in listeners; therefore, understanding the sentiment of music could provide valuable insights into the emotional characteristics and appeal of different songs. By applying sentiment analysis on music content, we could identify the emotional factors of tracks, such as the level of happiness, sadness, excitement, or relaxation. This information could be used to tailor recommendations based on the user's current emotional state or mood, providing a more personalized and engaging music experience.

Additionally, sentiment analysis could potentially assist in addressing the cold-start problem in recommendation, where limited user data is available. By analyzing the sentiment of music tracks, we could bridge the gap between user choice and music attributes, allowing for effective recommendations even for new users with sparse data.

By embedding sentiment into music recommendation, we aim at measuring the effectiveness of emotion, and the role it plays, in the representation of user preferences and the task of generating suitable candidates that cater for such preferences. Thus, we contribute to the development of music recommenders that not only consider user-item similarities or contextual information, but also incorporate the emotional aspects of music.

1.2. Goals

The primary objective of this project is to explore and understand the effectiveness of applying sentiment attributes in the domain of music recommendation, by investigating the role of mood in music choice and its representation of user preferences. This will be accomplished through the collection of a new dataset using Last.fm's Application Programming Interface (API), followed by the generation of sentiment attributes using a text sentiment analyzer. This dataset will be used to train different models and to test their integration of sentiment by evaluating recommendation accuracy. For this goal, context-aware models are of interest, as these algorithms are able to seamlessly embed additional attributes into the recommendation process. Finally, a web application will be developed as a platform to showcase the tools used throughout the project, along with the final recommendation models.

However, it is worth mentioning that the real value of the thesis lies in the process rather than the outcome since, throughout its making, it entailed the deepening of topics such as natural language processing, music recommendation and web scraping, which were not touched upon thus far; together with the opportunity to put into practice all the knowledge acquired during these years, namely software engineering, web development, database management and machine learning.

1.3. Work structure

This document is organized into five chapters, which can be summarized as follows:

Chapter 1. Introduction. Description of the topic around which the project revolves, the thesis motivation and work structure.

Chapter 2. State of the Art. Review of the concepts that serve as the theoretical foundation for the thesis, studying the latest advancements in recommender systems, music recommendation, natural language processing and web applications.

Chapter 3. Design and implementation. Definition of the design decisions made for each core component of the project, analyzing their functional and non-functional requirements, structure and lifecycle; along with an explanation of the technical aspects of the thesis, such as code architecture and implementation details.

Chapter 4. Experiments and results. Description of the tests conducted on the different models using the collected dataset, analyzing features and evaluating the settings and results obtained for each.

Chapter 5. Conclusions and future work. Overview of the conclusions drawn from the thesis, summary and discussion of the final results, as well as a review of potential improvements and areas for further research.

2

State of the Art

This chapter serves as an introduction and brief insight into the main concepts explained and discussed throughout the project, involving recommender systems focused on music, natural language processing and sentiment analysis, web applications and database management.

2.1. Recommender systems

Recommender systems are a type of intelligent systems designed to provide users with items of interest. This is usually achieved by building user profiles based on their preferences, past actions or information from their environment [1].

With the explosion of digital content and the growing complexity of user preferences, recommender systems have become indispensable for a satisfactory user experience, being present in industries such as e-commerce, social media, and more relevant to the matter at hand, entertainment, including music on platforms such as Spotify, SoundCloud or Last.fm. Recommendation of music, and media in general, is a particularly daunting task due to the variety of items and the subjectivity involved in every interaction, making it an interesting topic of research.

The latest developments in recommender systems encompass a range of techniques explained in the following sections, including collaborative filtering, content-based recommendation, hybrid systems, and deep learning methods. These techniques aim to provide accurate and personalized recommendations by making the most of user preferences, item characteristics, and contextual information.

2.1.1. Model classification

Recommender systems can be classified into different types based on their techniques and focus. These types include:

- **General Recommenders**, consisting of various techniques including collaborative filtering algorithms, which analyze user-item interactions to identify similar users or items, or **content-based** sys-

tems, which focus more on items' characteristics and attributes (e.g. category, language, author...), analyzing their content to suggest similar items according to the user's preferences. Non-personalized methods, such as popularity-based recommendation, also fall into this category [2].

- **Context-Aware Recommenders**, which incorporate contextual information, such as time, location, demographics, and user behavior, to provide personalized recommendations that adapt to the user's current situation. Additionally, these recommenders could also be **content-based**, incorporating relevant content as embeddings, for instance, to increase the amount of useful training data [3].
- **Sequential Recommenders**, which consider the temporal order of user interactions and exploit sequential patterns to make recommendations. They utilize historical sequences of user actions to predict the next item of interest [4, Chapter 3.5].
- **Knowledge-Based Recommenders**, which make use of external knowledge sources, such as knowledge graphs or semantic networks, to enhance the recommendation process. They utilize domain-specific knowledge and semantic relationships to generate more accurate and meaningful recommendations [5].

Hybrid approaches that combine multiple types are also common, allowing for leveraging different techniques to improve recommendations. The ultimate choice of model depends on the recommendation problem, available data and desired personalization and contextual relevance.

2.1.2. Context-aware recommendation

Context-aware recommender systems have garnered significant attention in recent years for their ability to take into account the specific context in which users interact while generating recommendations. Said context is comprised of contextual factors, which can be fully observable (explicit information), unobservable (the model extracts the latent variables implicitly) or partially observable.

Feature engineering

Observable factors might be obtained and processed through feature engineering, which plays a crucial role in context-aware recommendation. In particular, it involves selecting or creating relevant data that captures users' and items' characteristics or context through domain knowledge. These features may be tested and evaluated of their relevance and impact on the model's performance, some enhancing its accuracy, and some being more redundant.

Popular applications

Typically, this type of recommenders are not exclusively dependent on context, but instead incorporate methods to embed contextual information into the final model, some popular examples being:

- **Collaborative Filtering with Context**: By incorporating contextual information into the recommendation process, these systems extend traditional collaborative filtering and combine user-item preferences with contextual features to improve the model [3, Classical Approaches].
- **Matrix Factorization with Side Information**: Matrix factorization models are enhanced to incorporate contextual factors as additional input features. By jointly factorizing the user-item matrix and the context (or content) matrix, these models capture the influence of context on user preferences and generate contextually relevant recommendations. One popular application of this are Factorization Machines (FM) [3, Classical Approaches, Tensor Factorization].
- **Contextual Bandits**: Contextual bandit algorithms select items to recommend based on contextual information and observed user feedback. These algorithms aim to strike a balance between exploration and exploitation, considering the context to optimize recommendation choices over time [3, Reinforcement Learning].
- **Deep Learning with Context**: Deep Learning (DL) models, such as neural networks, can effectively learn complex patterns in contextual data. By integrating contextual information into the architecture, DL models capture the nuances of the user's preferences in different contexts [3, Deep Learning].

As it can be deduced, these type of recommenders perform best when hybridized with techniques that work on other aspects of the process or the data, in order to provide users with truly useful recommendations.

2.1.3. Music recommendation

Music recommendation is a specialized domain of recommender systems that focuses on providing personalized music suggestions to users. Modern day advancements on music recommendation involve sophisticated algorithms and machine learning models, as well as rich music data to enhance recommendation accuracy and user satisfaction [4].

Collaborative and Content-based filtering

Two popular techniques applied to music recommender systems are collaborative filtering and content-based filtering:

- **Collaborative Filtering**: This technique analyzes user behavior and preferences to identify similarities between users or items. It recommends music that is popular among users with similar tastes, enabling the discovery of new songs or artists according to collective preferences [4, Chapter 3.1].
- **Content-Based Filtering**: To find similar songs, this technique utilizes user-specific features, such as listening history, favorite genres or preferred artists; music-specific features, such as genre, tempo, acoustic properties or lyrics; and contextual features, such as time of day, location or mood. It tailors

recommendations characterized by the attributes of the music that the user has enjoyed, providing songs that align with their individual tastes [4, Chapter 3.2].

Both techniques are often centered around similarities between users or items and common interactions or characteristics, which can be obtained from metrics such as Euclidean distance or cosine similarity. In the context of recommendation, cosine similarity is a popular choice for content-based systems, as the content usually consists of tokens or pieces of text (e.g., summaries, comments, tags). First, however, these texts need to be vectorized, using methods like TF-IDF or count vectorization.

Cosine similarity

Cosine similarity is an effective measure for assessing text similarity in recommendation systems. It operates on vector representations of texts, capturing the semantic similarity between them by comparing the orientation of the vectors and their angle. It is robust to variations in text length, focusing on semantic content rather than the absolute magnitude of vectors, and also computationally efficient, making it suitable for large-scale datasets.

$$\cos(A, B) = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{|A| \cdot |B|} \tag{2.1}$$

Where A and B are vectors, with $|A|$ being the vector magnitude of A.

TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) [2.2c] is a text vectorization technique that assigns weights to words in a document, based on their importance in both the document and the corpus as a whole. It calculates a score by multiplying the Term Frequency (TF), which measures how frequently a term appears in a document, with the Inverse Document Frequency (IDF), which measures the rarity of a term across the entire corpus. By highlighting terms that are both frequent within a specific document and relatively rare in the overall corpus, it allows for the identification of more important and discriminative terms.

$$tf(t, d) = \begin{cases} 1 + \log freq(t, d), & \text{if } freq(t, d) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.2a}$$

$$idf(t) = \log \frac{|\mathcal{D}|}{|\mathcal{D}_t|} \tag{2.2b}$$

$$tf\text{-}idf(t, d) = tf(t, d) \cdot idf(t) \tag{2.2c}$$

Where $|\mathcal{D}|$ denotes the total number of documents in the corpus, and $|\mathcal{D}_t|$ the number of documents containing the term t.

Music datasets and APIs

To test and evaluate music recommendation models, researchers and developers often utilize popular datasets available in the field, such as the Million Song Dataset, which provides a vast collection of audio features and metadata for a million songs. Furthermore, platforms like Spotify and Last.fm offer APIs that allow access to music data, user listening history, and other relevant information, enabling researchers to experiment and assess their approaches with diverse and representative real-world data. In this project, sentiment attributes were extracted from Last.fm's tags associated with tracks, artists and albums, with the resulting dataset being evaluated on different models.

2.1.4. Evaluation

Evaluating recommender systems is essential to assess their utility and accuracy, and involves comparing the system's output recommendations with some ground truth data, such as user ratings, implicit feedback, or user interactions. Several evaluation metrics have been developed to measure different aspects of recommender system performance, tightly related to Information Retrieval (IR) evaluation.

Among them, some of the most popular metrics include Precision, Recall, F1 score, Mean Average Precision, Normalized Discounted Cumulative Gain and Mean Reciprocal Rank. Moreover, a cutoff is typically given to test with the top K results, since in the vast majority of cases the most interesting results are those shown first to the user.

Precision, Recall & F1 Score

Precision [2.3a] is a metric that measures the proportion of relevant items among the recommended items. It focuses on the accuracy of the system by calculating the ratio of true positives (relevant items recommended) to the total recommended items.

However, precision alone does not consider the number of relevant items that were not recommended, which is where Recall comes into play. Recall [2.3b] measures the proportion of relevant items that were actually recommended, providing insights into the system's coverage of relevant items. The F1 score [2.3c] combines Precision and Recall into a single metric, providing a balanced measure of both metrics.

$$P = \frac{|Relevant \cap Returned|}{|Returned|} \tag{2.3a}$$

$$R = \frac{|Relevant \cap Returned|}{|Relevant|} \tag{2.3b}$$

$$F_1 = \frac{2 \cdot PR}{P + R} \tag{2.3c}$$

Mean Average Precision

Mean Average Precision (mAP) is a widely used metric in recommender systems evaluation, especially in IR scenarios [2.4b]. It measures the Average Precision (AP) across different recall levels and is particularly useful when dealing with varying lengths of recommendation lists. mAP takes into account the position of relevant items in the ranked list of recommendations, penalizing systems that place relevant items lower in the list.

$$AP_k = \frac{1}{|Rel@k|} \sum_{k \in relevant} P@k \tag{2.4a}$$

$$mAP = \frac{1}{|Rel|} \sum_{i=1}^{|Rel|} AP_i \tag{2.4b}$$

Normalized Discounted Cumulative Gain

Normalized Discounted Cumulative Gain (NDCG) is another popular metric in IR or recommender systems evaluation [2.5c]. It considers both the relevance and the rank of recommended items, and stems from the normalization of the Discounted Cumulative Gain (DCG) [2.5a], with the ideally ordered DCG or IDCG [2.5b]. Higher scores are assigned to relevant items that are ranked higher in the recommendation list, applying a discount based on the position in the list. By considering the graded relevance of items, NDCG provides a more fine-grained evaluation of the system's performance.

$$DCG = \sum_{k=1}^{|Rel|} \frac{Relevance(d_k)}{\log_2(k+1)} \tag{2.5a}$$

$$IDCG = DCG_{Ideal\ order} \tag{2.5b}$$

$$NDCG = \frac{DCG}{IDCG} \in [0, 1] \tag{2.5c}$$

Mean Reciprocal Rank

Reciprocal Rank (RR) takes into account the rank of the first relevant item in the recommendation list [2.6a]. Likewise, the Mean Reciprocal Rank (MRR) averages the Reciprocal Rank from the results of several queries [2.6b]. These metrics are most useful to assess systems where the user requires just one relevant item from the recommendation list.

$$RR = \frac{1}{\text{mín}\{k \in relevant\}} \tag{2.6a}$$

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} RR_i \tag{2.6b}$$

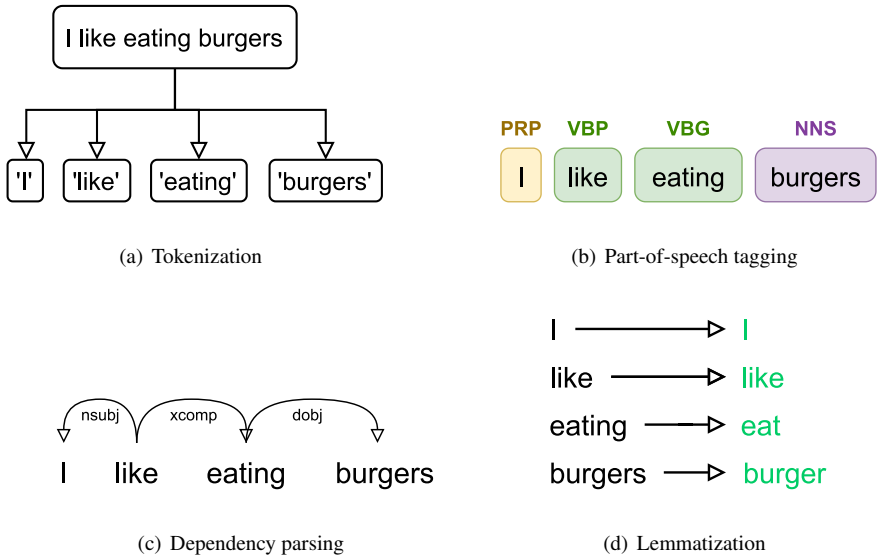
It is important to note that the choice of evaluation metrics depends on the specific application domain, the nature of the recommendation problem, and the available ground truth data. Different metrics provide insights into different aspects of the recommender system's performance, and multiple metrics are usually applied to obtain a comprehensive evaluation.

2.2. Natural Language Processing

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and human language. It encompasses a wide range of techniques and applications, enabling computers to understand, interpret, and generate human language text. Three important areas within NLP are text processing, sentiment analysis and word embeddings.

2.2.1. Text Processing

Text processing involves analyzing and manipulating textual data to extract valuable information. It includes tasks like tokenization, part-of-speech tagging, dependency parsing and lemmatization [2.1], which enable the extraction of insights and facilitate various applications such as sentiment analysis and information retrieval. Popular libraries have been developed that provide extensive functionalities for these tasks, such as spaCy [6] and Natural Language Toolkit (NLTK) [7], both used in this project.



(a) Tokenization

(b) Part-of-speech tagging

(c) Dependency parsing

(d) Lemmatization

Figure 2.1: Text processing techniques

a) **Tokenization** is the process of splitting text into smaller units called tokens, which can be individual words, phrases, or even characters — fundamental for subsequent analysis.

b) **Part-of-speech tagging**, or POS tagging, is the process of assigning grammatical tags to words in a text, indicating their syntactic category and role in a sentence. It helps in understanding the grammatical structure and meaning of sentences.

c) **Dependency parsing** analyzes the grammatical structure of a sentence by identifying the relationships between words and representing them as a hierarchical structure or dependency tree. It helps to uncover the syntactic dependencies between words and is typically useful for sentence segmentation and token dependency labelling.

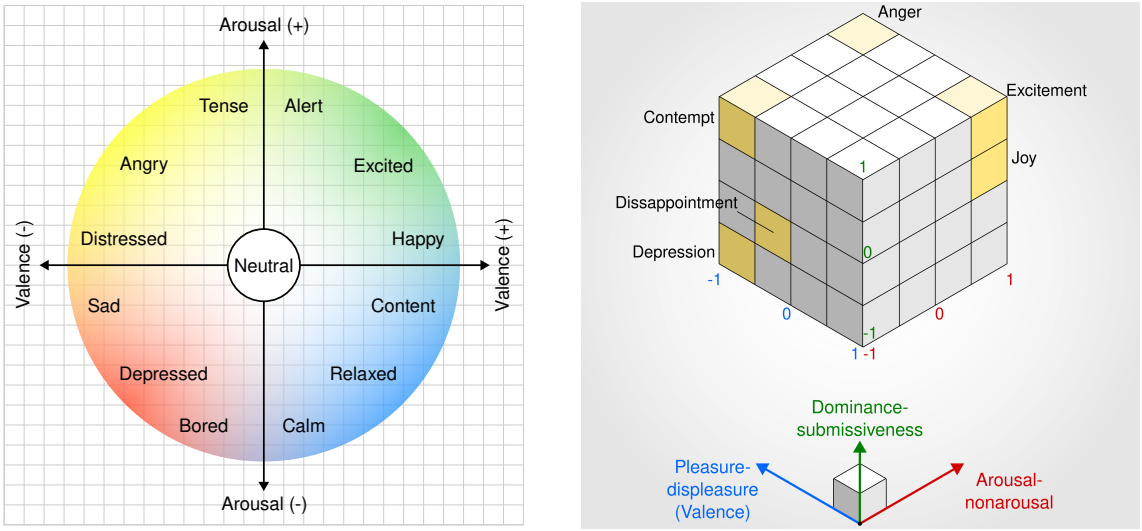
d) **Lemmatization** is the process of reducing words to their base or canonical form, known as a lemma, by considering their context and morphological features. It allows for better text analysis by treating different inflected forms of a word as a single normalized entity.

2.2.2. Sentiment Analysis

Sentiment analysis, also known as opinion mining, aims to determine the sentiment or subjective information expressed in a piece of text. It involves analyzing text data to identify the overall sentiment, such as positive, negative, or neutral, and sometimes even more nuanced emotions. One popular model used in sentiment analysis is the Valence-Arousal-Dominance (VAD) model [8].

VAD Sentiment Model

The traditional VAD model assigns scores to words based upon three dimensions: valence, arousal, and dominance. Valence represents the pleasantness or positivity of the sentiment, arousal represents the level of excitement or intensity, and dominance represents the degree of control or influence.



(a) Circumplex Valence-Arousal model

(b) Three-dimensional Valence-Arousal-Dominance model

Figure 2.2: Sentiment models based on valence, arousal & dominance, left representing the Valence-Arousal model [9], right representing the Valence-Arousal-Dominance model [10].

Frequently, valence and arousal have been considered sufficiently independent to convey mood and musical emotions [2.2(a)], since two dimensions are enough to represent a reliable sentiment spectrum, striking a balance between complexity and predictive potential; with dominance being necessary for a wider range of emotions [2.2(b)], at the expense of simplicity and scoring accuracy. These scores provide a more comprehensive understanding of the emotions conveyed in the text [11].

Sentiment Ratio

Another common approach in sentiment analysis is to calculate a sentiment score from the frequency of positive and negative words in the text [2.7]. This score can be computed by subtracting the number of negative words from the number of positive words and dividing it by the total number of words in the text. This technique provides a sentiment ratio, a quantitative measure of sentiment polarity that is akin to valence but ignoring neutral words [12].

$$Sentiment\ Ratio = \frac{|Positive\ words| - |Negative\ words|}{|Total\ words|} \in [-1, 1] \tag{2.7}$$

2.2.3. Word Embeddings

Word embeddings are a representation of words in a continuous vector space, where words with similar meanings are placed closer together. Word embedding models, such as Word2Vec [13] and GloVe [14], learn these vector representations by considering the context in which words appear in large amounts of text data. These representations capture semantic relationships between words and enable various downstream NLP tasks to benefit from this contextual understanding.

Word embeddings have proven to be effective in a wide range of NLP applications, including sentiment analysis, named entity recognition, machine translation, and document classification. They enable machines to capture the meaning and context of words, enhancing their ability to understand and generate human language.

The NRC-VAD lexicon, used in this project, is a collection of over 20,000 words to which fine-grained VAD scores were manually assigned, in the form of emotion-aware word embeddings. Table 2.1 shows the highest and lowest scored words for each dimension, fitting the descriptions from Section 2.2.2 with fair reliability [15].

Dimension	Word	Score ↑	Word	Score ↓
valence	*love*	1.000	*toxic*	0.008
	happy	1.000	*nightmare*	0.005
	happily	1.000	*shit*	0.000
arousal	*abduction*	0.990	*mellow*	0.069
	exorcism	0.980	*siesta*	0.046
	homicide	0.973	*napping*	0.046
dominance	*powerful*	0.991	*empty*	0.081
	leadership	0.983	*frail*	0.069
	success	0.981	*weak*	0.045

Table 2.1: NRC-VAD lexicon entries showcasing words with the highest and lowest scores for each of the three dimensions.

2.3. Web applications

With the aim of showcasing the tools used throughout the data acquirement process, as well as providing a user-friendly testing experience on the better models, a web application has been developed using popular toolkits and frameworks, namely SQLAlchemy and Django.

Web applications are typically comprised of three core components:

- The **backend**, often powered by frameworks like Django, handles the business logic, user requests processing, database communication. It manages the application's functionality, authentication, security, and API endpoints.
- The **frontend**, on the other hand, focuses on the user interface, utilizing technologies such as HTML, CSS, JavaScript and, in the case of Django, the Jinja2 template engine, to create visually appealing and interactive experiences.
- The **database**, powered by management systems like PostgreSQL or MySQL, stores and manipulates the application's data, providing efficient data storage, retrieval, and management.

2.3.1. Django Framework

In the latest years, several frameworks have gained popularity due to their ease of use and robust features, with Django being one of the most prominent [16, 17]. Django is a web framework built on Python, renowned for its simplicity, scalability, and rapid development capabilities. It provides a comprehensive set of tools and libraries that simplify the development process, including an Object-Relational Mapping (ORM) for database management, a powerful templating engine for user interface rendering, and a routing system for handling URL requests. Django follows the Model-View-Template (MVT) software design pattern, an extension of the traditional Model-View-Controller (MVC) pattern.

MVT design pattern

The Model-View-Template pattern separates the different aspects of a web application, providing a structured approach for handling data (Model), presentation logic (View), and user interface (Template).

- The **Model** interacts with the database. It defines the data structure and provides methods for querying, updating, and manipulating the database. When a URL is accessed, it is responsible for retrieving relevant data from the database or updating existing data depending on the request.
- The **View** handles the logic behind processing the user's request. After receiving the necessary parameters from the URL request, it interacts with the Model to retrieve or modify data as needed, and may perform additional operations based on the request parameters. Finally, the View prepares the data to be passed to the Template for rendering.

- The **Template** is responsible for handling the user interface. It defines the structure, layout, and presentation of the page, incorporating any necessary data from the View to populate the page with the requested information. Once the Template has processed the data, it generates the HTML response to be sent back to the user's browser.

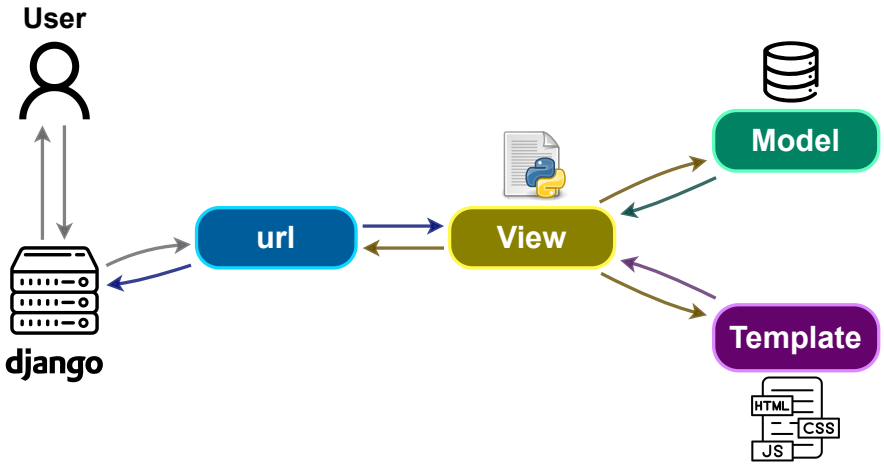


Figure 2.3: Django's Model-View-Template architecture

When users access a webpage built on Django, all pattern components work together to handle the requests and generate the corresponding responses, as depicted in Figure 2.3. This division of responsibilities allows for a clean separation of concerns and promotes code modularity and maintainability, making it easier to modify or update specific components without affecting the others.

2.3.2. Database Management

Efficient database management is a critical aspect of web application development, and Django excels in this area. It offers seamless integration with various databases, including PostgreSQL, a powerful open-source relational database management system.

Relational databases are known for their reliability, robustness, data integrity and Structured Query Language (SQL), and are inherently supported by Django as its primary way of persisting the data and relationships between tables.

Django's built-in ORM layer simplifies database interactions by allowing developers to work with databases using models defined in Python, which serve as blueprints for creating database tables and managing data. Nevertheless, when the amount of data is too large and querying speed is paramount, Django's ORM might be outperformed by other libraries such as SQLAlchemy.

SQLAlchemy Toolkit

SQLAlchemy is a popular Python library for advanced database operations, which offers a higher level of abstraction and flexibility when working with database systems, including PostgreSQL, which was used in this project. It provides a powerful toolkit for creating complex queries, performing data manipulations and optimizing database interactions [18]. While not an integral part of Django, SQLAlchemy can be seamlessly embedded into Django projects, to provide a more versatile, efficient and scalable database management when needed.

In addition, this toolkit includes its own ORM, which abstracts away the complexities of raw SQL queries and allows developers to focus on application logic and development speed. Being built in Python, SQLAlchemy provides additional flexibility and capabilities for customizing and parameterizing queries.

Code 2.1: Example usage of SQLAlchemy's ORM interacting with an existing database. Function *get_user_id* filters a user by the given username, returning their ID.

```
3  db = sqlalchemy.create_engine("postgresql://alumnodb:alumnodb@localhost:5432/lastfm_db",
4                                client_encoding="UTF-8")
5
6  # Load table definitions from DDBB
7  metadata = sqlalchemy.MetaData()
8  metadata.reflect(bind=db)
9
10 # Mapping for table 'user'
11 USER = metadata.tables['user']
12
13 def get_user_id(username: str):
14     stmt = sqlalchemy.select(USER.c.id).filter(USER.c.username == username)
15     result = db.execute(stmt).first()
16     return result
```

For many projects, Django's ORM provides ample functionality, eliminating the need for an additional library like SQLAlchemy. The choice between Django's ORM and SQLAlchemy ultimately depends on the database requirements and complexity of the project, which is accounted for in this case.

3

Design and Implementation

This chapter presents a comprehensive overview of the project's key components, encompassing their structure, requirements analysis, design decisions, and implementation details.

3.1. Project structure

The project is divided into three core modules:

- *Web application*. System that provides a user interface to illustrate the tools used thoughout the project, as well as the final recommendation models.
  - *Recommendation models*. Trained on the collected dataset, they were provided by *RecBole*, a library which will be described in following sections.
- *Dataset*. Data scraped from Last.fm and complemented with sentiment analysis, in order to have the experimental data for model training. This chapter will detail the process of obtaining such dataset.
  - *Data scraper*. Script that handles data collection through API calls, web scraping from Last.fm, and generation of sentiment attributes.
  - *Sentiment analyzer*. Script that analyzes textual content and generates sentiment scores, using the NRC-VAD lexicon and optimized NLP libraries such as spaCy and NLTK.
- *Database*. Storage system into which the dataset is inserted, in order to maintain an organized, stable and efficient way of accessing the data used by the recommendation models.

The web application serves as a link between all modules due to the tools interacting actively with each one, as outlined in Figure 3.1, which are explained in more detail in Section 3.3.1. Even though dataset acquirement is a one-time procedure, some intermediate steps, specifically user data scraping and sentiment analysis, are showcased as well.

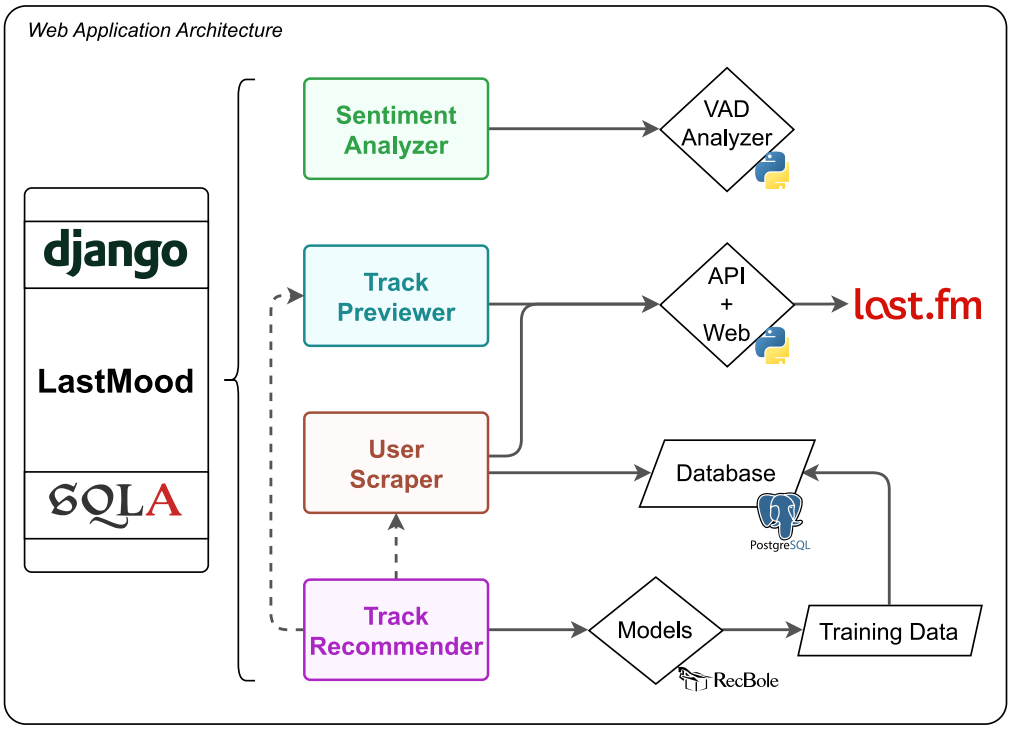


Figure 3.1: Web application architecture

3.2. Requirements analysis

In this section, the requirements for each module and sub-module of the project will be identified and documented, classifying them as functional (what the component should do) and non-functional (quality attributes and constraints) in order to outline their purpose and guide the development process.

3.2.1. Web application

Functional requirements

FR-WA-1.– The web application will provide a Last.fm track previewer that allows users to listen to music samples from YouTube and Spotify.

FR-WA-2.– The web application will include a text sentiment analyzer that receives text inputs and presents analysis results in a table format, including the extracted sentiment attributes.

FR-WA-3.– A Last.fm user scraper will be part of the showcase tools, allowing users to choose the type and quantity of data to be scraped (tracks, artists, albums, tags), specify the username, and select the data source (API or scraped database). The scraped results will be displayed using tables or piecharts to allow easy analysis.

FR-WA-4.– The web application will be able to generate music recommendations using the best models trained on the scraped dataset. Users should be able to select the desired model, set the username, results cutoff, and define the number of results per page.

FR-WA-4.1.– If no username is selected and the model is personalized, a random user will be sampled from the database.

FR-WA-4.2.– Each recommended track will include details such as artist, album, tags, rank and VAD values, including a link to its preview page using the Last.fm track previewer.

FR-WA-4.3.– The recommendation results will include a link to the Last.fm user scraper endpoint for the chosen (or random) user, allowing to compare their data with the recommended tracks.

Recommendation library and models

FR-WA-5.– The recommendation models must be deployable on the web application, ensuring adequate prediction times when users request recommendations.

FR-WA-6.– The recommendation models must be serializable, as no training should be done for deployed models.

FR-WA-7.– The recommendation models must be capable of providing personalized recommendations for any user in the dataset, by means of the chosen user's data.

FR-WA-8.– The context-aware recommendation models must allow incorporation of additional features into the training process, automatically or through configuration settings.

FR-WA-9.– The recommendation library must provide a way to implement custom models, through abstract recommender modules, for instance.

Non-functional requirements

NFR-WA-1.– The web application should provide a responsive and intuitive user interface for seamless interaction with the platform's features.

NFR-WA-2.– The system should be scalable to handle a growing number of users and accommodate future updates and enhancements.

NFR-WA-3.– The application should have low latency, ensuring quick response times to user requests; otherwise, an indication should be shown.

NFR-WA-4.– The recommendations generated by the web application should have a user-friendly interface to browse the recommended tracks.

Recommendation library and models

NFR-WA-5.– The recommendation models should be implemented as independent code modules, to enable easy maintainability and expansion, allowing for any additional functionality or customization of training processes.

NFR-WA-6.– The recommendation library should offer flexible and configurable options to select data features and settings for training and evaluation.

NFR-WA-7.– The recommendation models should be optimized for efficient training and prediction, using high-performance computing techniques such as CPU vectorization or GPU parallel computing.

3.2.2. Dataset

Functional requirements

FR-DS-1.– The dataset will be static, meaning that no new data will be added at any point, as it should be stable to provide consistent results for evaluation purposes.

FR-DS-2.– The dataset must include data from a minimum of 50,000 users to ensure a substantial user base, necessary for a reasonable evaluation of the recommendation models.

FR-DS-2.1.– Each user will have associated recent, top and loved tracks, as well as top artists and

top albums, to capture diverse preferences and generate more extensive user profiles.

FR-DS-3.– The dataset will include tags associated with each track to perform sentiment analysis on.

FR-DS-4.– The dataset will include additional information about tracks, namely artists and albums, to enable a fair comparison between the embedding of sentiment attributes and other track features.

FR-DS-4.1.– Artists and albums will include their own associated tags to expand on tag data, contributing to track tagging and reduction of data sparsity.

Sentiment analyzer

FR-DS-5.– The sentiment analyzer will generate sentiment attributes, specifically valence, arousal, dominance, and sentiment ratio or proportion of positive to negative words, for any given English text.

FR-DS-6.– The sentiment analyzer must load all necessary information into memory during initialization, which should occur only once upon module importation, to optimize subsequent analysis.

FR-DS-7.– The sentiment analyzer must support two modes of text processing: whole text and sentence-segmented; allowing analysis at both document and sentence levels.

FR-DS-8.– The sentiment analyzer must allow the final sentiment attributes to be obtained from the mean or the median of the individual scores.

Non-functional requirements

NFR-DS-1.– The dataset acquirement should be reasonable in both time and size, taking into account API rate limits, computer constraints and time dedication.

Sentiment analyzer

NFR-DS-2.– The sentiment analyzer should use optimized, state-of-the-art libraries for text processing, so its usage is viable for real world applications.

3.2.3. Database

Functional requirements

FR-DB-1.– The database will store a table for each main entity of the dataset: users, tracks, artists, albums and tags; with their respective identifiers and names.

FR-DB-2.– The database will store multiple tracks, top artists and top albums associated with one or more users, establishing many-to-many relationships stored in independent tables.

FR-DB-2.1.– Each track must have an associated artist, corresponding to a one-to-many relationship between artists and tracks.

FR-DB-2.2.– Each track could belong to an album, establishing a one-to-many relationship between albums and tracks.

FR-DB-2.3.– Each album must have an artist, establishing a one-to-many relationship between artists and albums.

FR-DB-3.– The database will store tags associated with tracks, artists and albums, requiring separate tables and appropriate references to store these many-to-many relationships.

3.3. Design

In this section, the design of the web application, dataset acquirement and database will be defined, providing details on their architecture, components, intercommunication between each other, and processes involved, by use of explanatory diagrams.

3.3.1. Web application

The web application, named *LastMood*, was built to showcase the tools used during data collection and the final recommendation models trained on said data, by providing a user-friendly interface. For this purpose, it plays the role of the central link that unifies every component, as a representative of the work done throughout the project.

As it can be seen in Figure 3.1, the application consists of four tools: the Track Previewer, the Sentiment (VAD) Analyzer, the User Scraper and the Track Recommender. These will be explained with sequence diagrams, to visually depict the interactions between each component and the flow of actions taken by the system.

Track Previewer

The **Track Previewer** is designed to allow users to request a track by specifying the artist and title, and retrieve relevant information and previews from Last.fm. As noted on the sequence diagram in Figure 3.4, the previewer makes use of both Last.fm's API and webpage to gather the necessary data.

1. First off, users provide the artist and title of the track they want to preview, which must exist in Last.fm. The Track Previewer then makes calls to Last.fm's API to obtain the context of the track, including the artist and track URL.
2. Afterwards, the component accesses the track's webpage by sending a request to the URL, and scrapes the HTML content to search for the containers that hold the Youtube and Spotify previews from Last.fm. Upon finding them, the scraper aims at the preview identifiers, such as the Youtube video ID or Spotify track ID.
3. Finally, the template for the previewer inserts the extracted previews into the webpage, as media embeds. Upon any error, such as non-existent tracks or previews missing from the webpage, the template dynamically reflects the error as feedback to the user.

The main use of this tool is for the track recommender to include usable previews in the recommendations, enabling easier interaction and comparison of tracks listened by the users in the database.

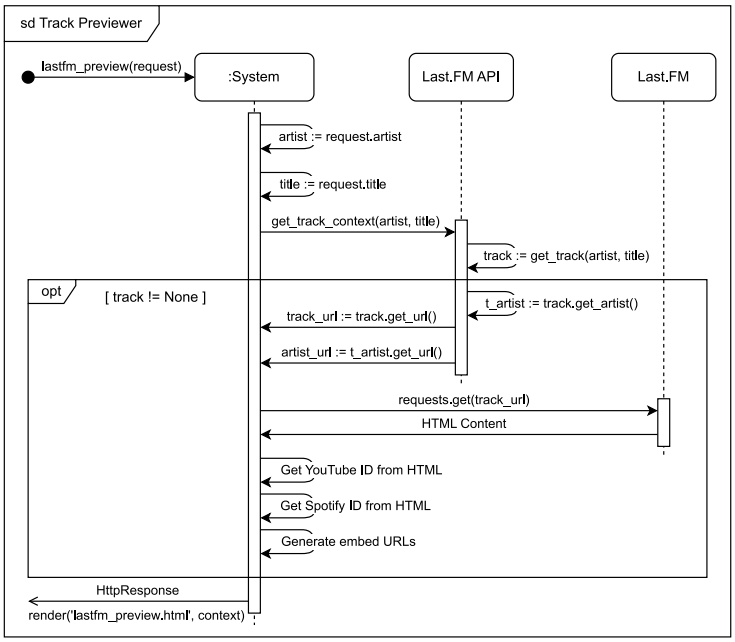


Figure 3.2: Sequence diagram: Track Previewer

VAD Analyzer

As observed in Figure 3.3, the **VAD Analyzer** requests an input text and two options: the computation mode for the VAD scores by mean or median of the text, and the analysis method for the text, as a whole or divided by sentences. The analysis results are presented in a table format, where each row represents either the entire text or individual sentences. The columns in the table include the text itself, along with valence, arousal, and dominance scores. Additionally, each row includes a sentiment label (positive, negative, or neutral), the sentiment ratio, the number of words analyzed, and a list of the words as lemmas. The analyzer was used during the dataset collection to generate sentiment attributes, and its functionality details are explained in Subsection 3.4.2.

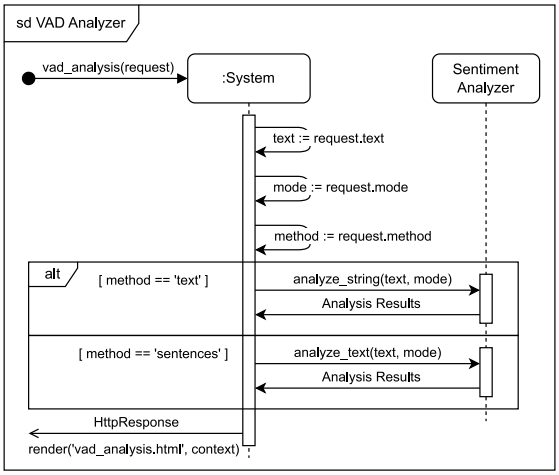


Figure 3.3: Sequence diagram: VAD Analyzer

User Scraper

The **User Scraper** tool in the web application allows users to input a username and select options, such as whether to retrieve data from the database or Last.fm's API, or enabling scraping and setting result limits for tracks, artists, albums, and tags, as represented in Figure 3.4.

For tracks, the scraper can obtain three types: top, loved and recent tracks; each including its title, artist and album. Recent tracks also have a "listened at" column, which captures the timestamp of when the user listened to the track, while loved tracks have a "loved at" column indicating the moment when the track was liked. Artists are stored with their names, and albums are associated with their titles and corresponding artists. At the end of the scraping process, and if the tag option is enabled, tags are extracted using the previous items and appended to their information.

All scraped items are organized into tables for easy visualization, in addition to a pie chart representing the frequency of each tag across all items, allowing to identify the most common tags listened by the user. This tool showcases part of the functionality developed for the dataset scraper, responsible for the acquirement of the dataset used to obtain sentiment attributes and train the recommendation models, and the implementation of which is explained in Subsection 3.4.2.

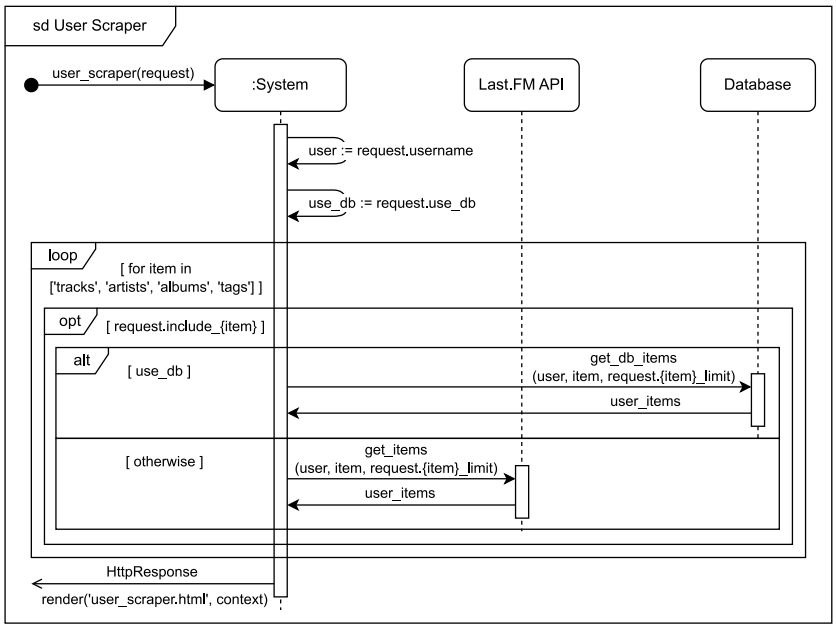


Figure 3.4: Sequence diagram: User Scraper

On another front, the track recommender generates links towards this tool, gathering data from selected users and providing an interface to compare recommendation results with a user's preferences and listen history.

Track Recommender

The **Track Recommender** offers users the ability to generate recommendations from one of the best recommender models, which were trained and evaluated with different metrics and features (see Chapter 4). As outlined in Figure 3.5, users can input a username from a list of available usernames in the database, or leave it blank to select one at random. They can also specify a cutoff for the top results to display, and set a limit for the number of results per page, providing pagination functionality. Some recommenders may have additional parameters which would be shown when the model is selected.

After requesting the recommendations, the system loads the trained model, which then returns the predicted tracks and scores for the chosen user. Then, the relevant context for each track is obtained from the database and finally returned as a catalogue, where each track is displayed in a card format. The cards contain information such as the track title, artist, album, rank, and associated tags. Users can have access to the track's preview page, or to a modal that shows additional details, including the tags of the track, artist, and album, as well as their VAD and sentiment score. The result page also includes a link leading to the user scraper page for the chosen user, as a way of viewing their listen profile.

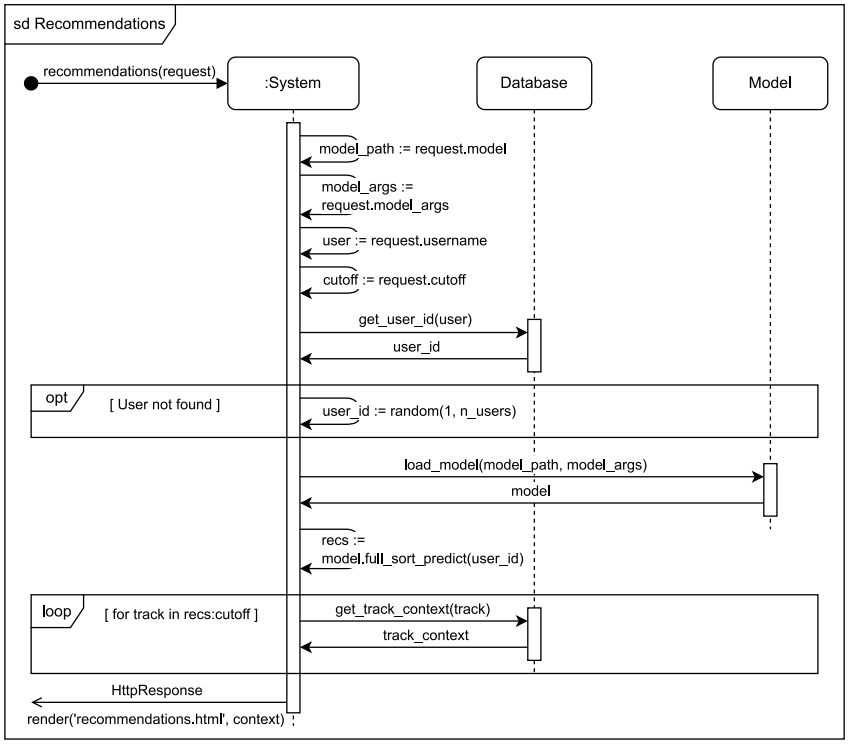


Figure 3.5: Sequence diagram: Recommendations

Recommendation models

The web application utilizes RecBole [19], a comprehensive and efficient recommendation library that provides several modern recommendation models, as well as a framework to work with these models. The decision to use RecBole over other libraries is detailed in Section 4.3.2.

RecBole offers various utilities for training and evaluation, including implemented metrics, options for data loading, processing and sampling, and automatic parameter tuning tools. The library allows for easy customization through provided interfaces and supports configuration through files and parameters (see Appendix B for an overview of the framework).

In terms of model design, RecBole includes an *AbstractRecommender* class, which serves as the base for four types of recommenders, namely *GeneralRecommender* (general algorithms), *SequentialRecommender* (next-item prediction), *KnowledgeRecommender* (knowledge-based), and *ContextRecommender* (context-aware), each being abstract classes themselves, as depicted in the class diagram of Figure 3.6. RecBole offers several models for each recommender type, which need to define methods for calculating training loss and predicting user-item interactions.

Three more models were designed as a means to generate additional feedback for this project (see Appendix D for implementations):

- **RandomRecommender**, a general recommender which generates random scores for each track, and may be used as a baseline model.
- **CosineSimilarityRecommender**, another general recommender which leverages the tags associated with tracks to compute similarities using TF-IDF vectorization. Recommendations are therefore based on tag similarity, and additional parameters can be adjusted to assign weight to tag ranking.
- **HybridVADRecommender**, a *ContextRecommender* that inherits from other context-aware models. It calculates scores by averaging the scores generated by the inherited model, and the Euclidean distance between the VAD values of tracks. The closer the tracks align with a user's average VAD, the higher the score.

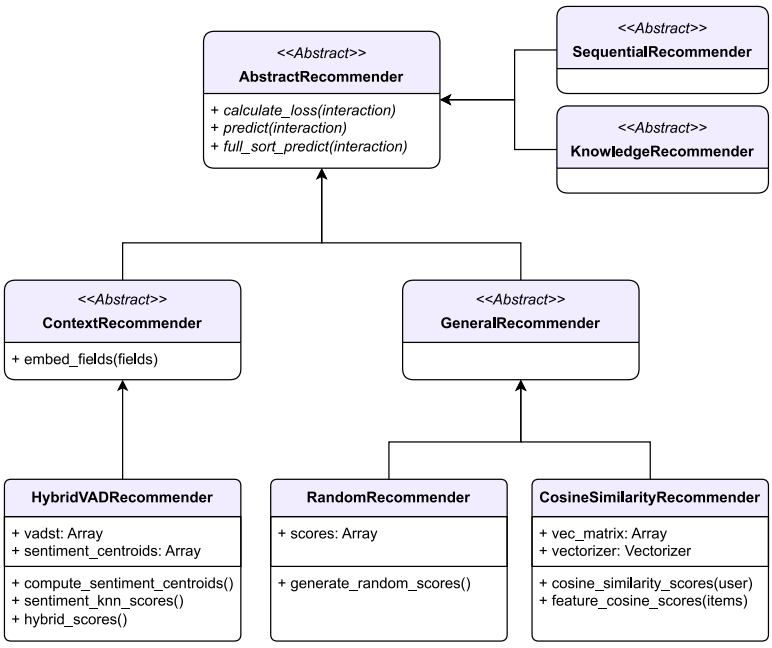


Figure 3.6: Class diagram: Recommendation models

3.3.2. Dataset

With the intention of evaluating sentiment applied to music recommendation, a suitable dataset is required, which must contain relevant data needed for recommendation models, such as user-item interactions and user or item features. Nevertheless, it must also include sentiment attributes to be able to evaluate the performance of sentiment-aware recommenders, for which a sentiment analyzer needs to be used.

Therefore, it is paramount to define a proper design for the dataset collection, which consists of two modules: the data scraper responsible for obtaining said dataset, and the sentiment analyzer used to extract sentiment scores from textual content — the detailed implementation of both being provided in later sections, taking into consideration the requirements from Section 3.2.2:

The **data scraper** is designed to acquire relevant information from Last.fm's API and webpage, with the objective of obtaining a considerable dataset to test recommendation models on. This collection process was necessary since no other publicly available dataset included users, tracks, artists, albums and tags; all of which are required for data processing, sentiment analysis and recommendation.

The scraper needs to access the API and obtain the data in a manner similar to web crawlers, in that the scraped items should give access to new items, valuable for the tasks at hand. The general procedure, as outlined in Figure 3.7, will consist of obtaining all the tracks, artists and albums along with their interactions, then getting their associated tags, and finally extracting sentiment attributes from textual content related to those tags. All of this data will be saved into files at first, and when the scraping is done, it will be inserted into an appropriate database.

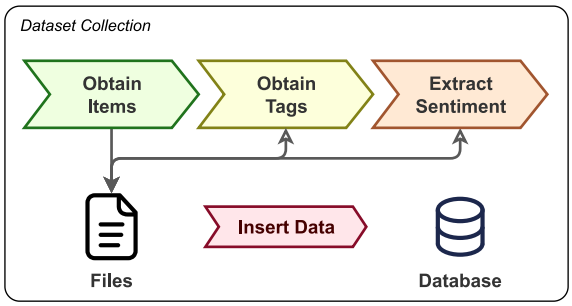


Figure 3.7: Dataset collection

The **sentiment analyzer** will provide an efficient way to extract VAD scores from any given English text. It was required to implement a new analyzer due to the lack of public libraries that focused on VAD scores, as the existing ones either were too simple or did not use the NRC-VAD lexicon, which is currently the most complete and reliable lexicon for these specific scores.

The analyzer will be imported into the data scraper, providing functions that allow it to perform sentiment analysis in a transparent way. To demonstrate its capabilities, the web application will also include the analyzer as part of the tools used in the project.

3.3.3. Database

The Entity-Relationship diagram portrayed in Figure 3.8 represents the database into which Last.fm's scraped data was inserted, designed to access the data in a standard and efficient manner. It is comprised of the five main entities (User, Artist, Track, Album, Tag) and all the interactions amongst them.

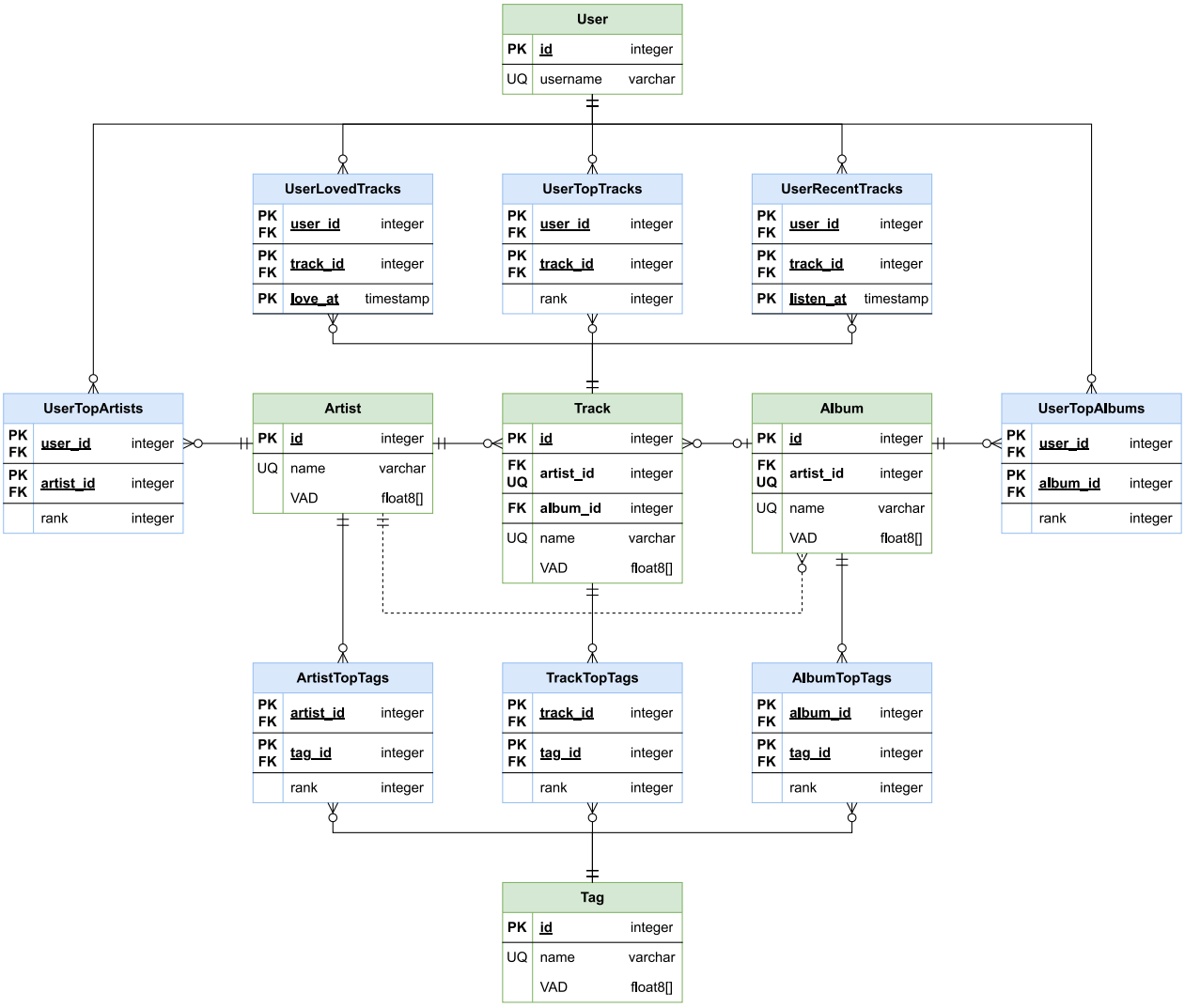


Figure 3.8: Entity-Relationship diagram of the Last.fm scraped database, where PK, FK and UQ mean primary key, foreign key & unique, respectively.

Following the diagram, all entities have tags (and thereby VAD scores) except users, since the API could not scrape personal tags from users other than oneself. These tags are user-based, meaning they are assigned by Last.fm users, with the assignment frequency determining their rank of representation.

Furthermore, all users have top items with which they interacted the most: artists, albums and tracks; along with loved and recent tracks. Loved tracks are those personally tagged by the user as one of their favorites, marked by the *love_at* timestamp, whereas recent tracks are the latest tracks the user has listened to up to the date of scraping, marked by the *listen_at* timestamp.

3.4. Implementation

This section provides an overview of the implementation details of the modules described in Section 3.1. It covers the external libraries used, followed procedures, and programming considerations.

3.4.1. Web application

For the development of the web application, the package of choice was the Python-based web framework Django, owing to its extensive feature set and robust MVT architecture, which facilitated the development process. Python, being the main programming language used throughout the entirety of the project, was an ideal choice for the backend functionality due to its readability and extensive support for machine learning and data analysis libraries, such as Pytorch [20], TensorFlow [21] and Pandas [22].

Additional technologies were used for the frontend as well, such as Bootstrap 5, jQuery and AJAX, which make the application feel responsive and dynamic. For instance, the user scraper makes asynchronous, independent AJAX calls to the server for each of the scraped items, such that the user may freely scroll and view the data obtained at that moment without having to wait for the entire process.

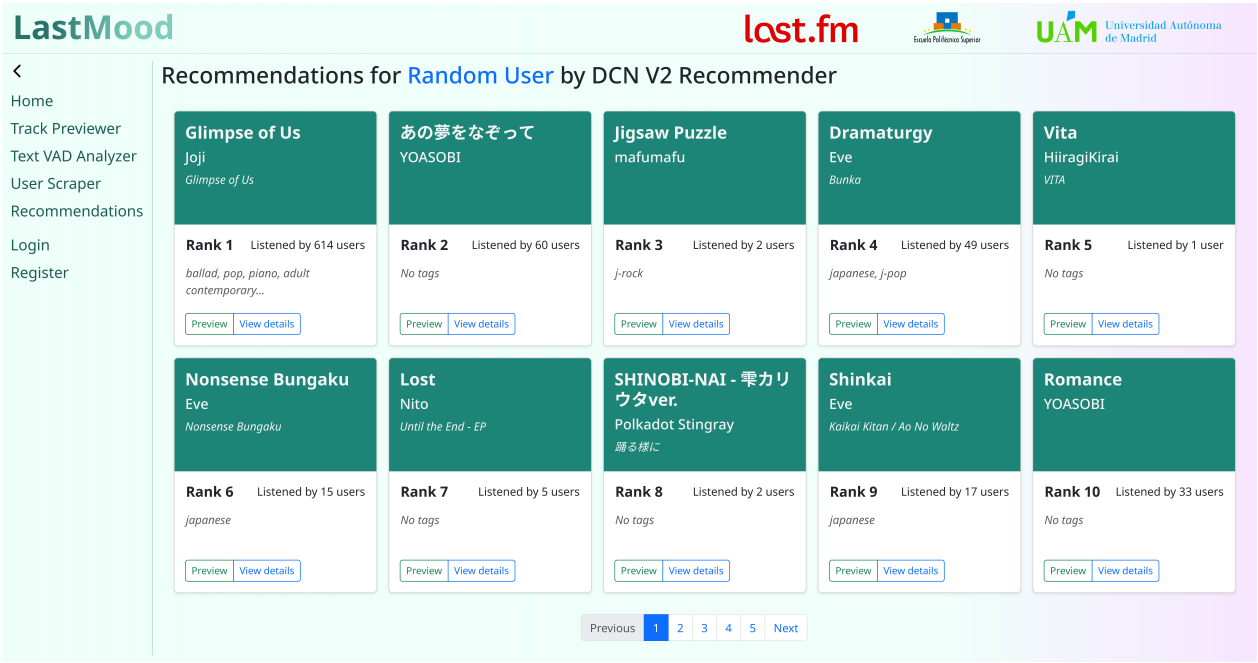


Figure 3.9: Example music recommendations (see Appendix A for additional screenshots).

Figure 3.9 presents example recommendations generated for a random user, arranged as a catalogue of tracks. This was possible thanks to Bootstrap's grid layouts, styles and modals.

3.4.2. Dataset

Data scraping from Last.fm

The implementation of the data scraper involves the use of several libraries such as pylast, a Python wrapper for Last.fm's API [23], BeautifulSoup for parsing HTML content from the webpage, and JSON for serializing the raw data. All API accesses by any module of the project were done through pylast, as it provides useful classes and methods to acquire mostly all the needed data.

One crucial aspect of the implementation is the management of network and data exceptions. To handle potential failures, the scraper incorporates a retry mechanism where, in case of an exception, the program retries the operation a certain number of times before moving on. Considering the substantial duration of the entire scraping process, this approach ensures that the scraper can recover from transient network issues or other exceptions and continue without being entirely hindered by individual failures.

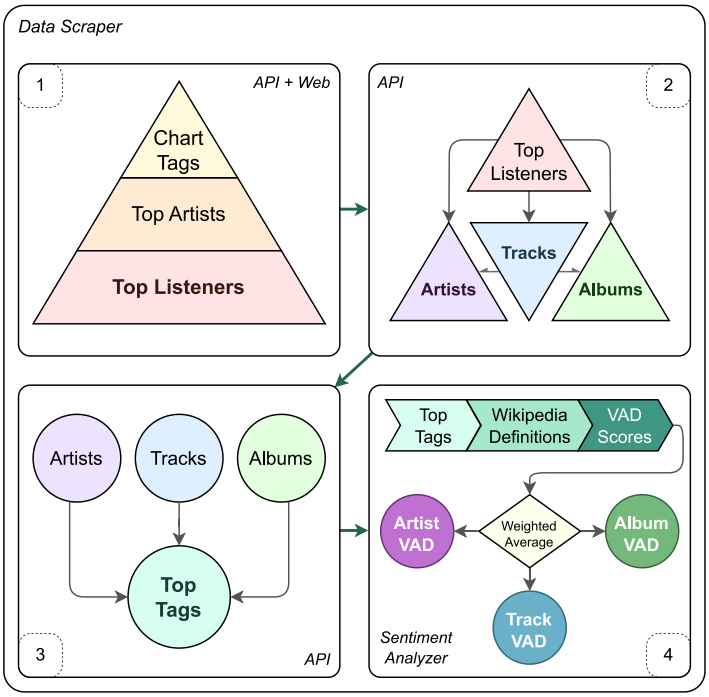


Figure 3.10: Steps of data scraping

The scraping is performed in various steps, which can be executed separately but depend on the data from previous ones, as represented in Figure 3.10:

1. First of all, the data scraper retrieves the top 50 chart tags using the API, representing the most listened tags at the time. For each tag, the scraper obtains the top unique artists associated with them, navigates to the artists' webpages on Last.fm and collects the top unique listeners, considering a maximum of 30 for each artist. These listeners need to be stored as they will represent users in the database, forming the basis for the recommendation models.

2. Next, the scraper utilizes the API to gather data from the top listeners, acquiring their top 20 tracks, recent tracks, and loved tracks, each with the corresponding timestamp, artist and album information. Additionally, it collects the top 10 artists and albums (with their respective artists) for each listener. All of this information is then stored as potential user-item interactions and features for the recommendation models.
3. The scraper continues by fetching the top 10 tags assigned by users to each unique track, artist, and album; before storing both the unique tags and all the item-tag assignments, which will be used in the following step.
4. Lastly, to complement the items with sentiment attributes, the scraper retrieves definition summaries for each unique tag using a Python implementation for Wikipedia's API [24], and then employs the developed sentiment analyzer over the texts to extract VAD and sentiment scores. For each unique track, artist, and album, weighted averages of the sentiment attributes are calculated based on the associated tags. The importance of each tag is determined by its rank, with the higher-ranking tags carrying much more weight as they are the most representative.

The data in raw files bears little use, so it must be inserted into a suitable database, as designed in Section 3.3.3. Also, a clear flaw in the process is the text used for sentiment analysis, as definitions are typically objective, and may not be sufficiently representative of the tags. However, it was the only source of reliable, standardized text that could still produce substantial and useful attributes.

Sentiment analyzer

The sentiment analyzer makes use of a custom spaCy NLP pipeline (see Appendix B) that takes care of all the text processing via pipes, which are essentially trained models that focus on specific steps and techniques. In this case, the pipeline was optimized to disable unnecessary pipes and use only the minimal and fastest versions for each step. The analyzer also leverages NLTK's WordNet, a corpus reader that includes synsets or synonym sets, which allow to search for words of similar meaning derived from their lemma and POS tag.

Considering the diagram in Figure 3.11, we can divide the analysis in the following steps:

1. Upon execution, the analyzer initializes the necessary resources, including loading the NRC-VAD lexicon and configuring the spaCy NLP model and pipeline. When a piece of text needs to be analyzed, the NLP pipeline parses and divides it into tokens, from which the POS tags and lemmas are obtained.
2. Afterwards, sentiment analysis is performed by iterating through the tokens, filtering out non-alphabetic and stop words, searching for the lemmas in the NRC-VAD lexicon and extracting sentiment values. To increase analysis coverage, the analyzer incorporates synonym search functionality. When a word is not found in the lexicon, the analyzer makes use of synsets to retrieve words related to the ori-

ginal lemma, effectively expanding the search scope and providing a broader analysis of sentiment nuances.

3. When a token is found, the analyzer includes procedures to handle sentiment modifiers, such as negation and degree adverbs. It detects and considers the positioning of such modifiers in the analyzed text, enabling adjustments to sentiment scores depending on the POS tags and whether they are negating, increasing or decreasing. After applying the score modifications, the final valence is used to assign a sentiment label: positive, negative or neutral.

4. Finally, after processing all tokens in the text, the analyzer then generates the final VAD scores by computing either the mean or median, as well as an additional sentiment ratio by dividing the difference between positively and negatively labeled words, by the total number of tokens found in the sentence or overall text.

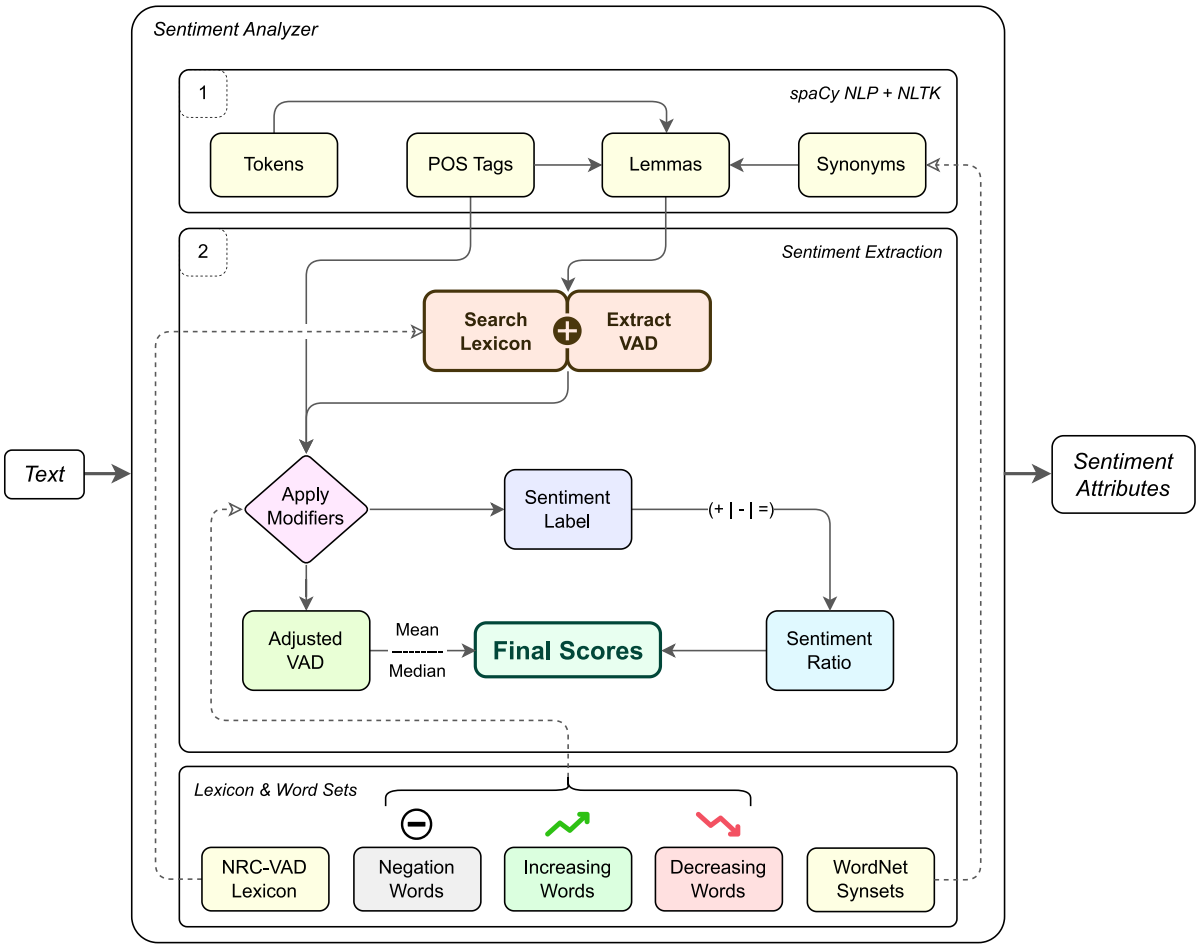


Figure 3.11: Sentiment analyzer structure

This sentiment analyzer was inspired from a simpler version implemented with NLTK and StanfordNLP, a different natural language processing package; which made use of a lexicon smaller than NRC-VAD [25]. The inclusion of sentiment modifiers was also incluenced by VADER, a sentiment analysis tool that offers accurate computation of valence in the context of social media [26].

3.4.3. Database

As explained in Section 2.3.2, despite the added complexity when compared with Django's ORM, SQLAlchemy was the library of choice to perform the necessary data retrieval operations from Python, as its ORM offers more advanced and versatile querying functionality, in addition to higher performance when combining tables for filtered results. The database was laid out with bigER Modeling Tool [27], which provided a visual interface to draw precise Entity-Relationship diagrams, and offered automatic code generation to create the designed tables.

The population of the database with the scraped data was implemented with SQLAlchemy as well, reading from raw files and generating insertion queries through the ORM, first for all the individual entities, and subsequently for every user-item and item-tag interaction. Since most of these interactions present similar relationships in the database, the code was written to avoid redundancy as much as possible. In order to ensure item uniqueness and standard encoding, each of the tracks, artists, albums and tags were filtered out to avoid repetitions or invalid characters.

After data insertion, the resulting dataset can be summarized by the following record counts displayed in Table 3.1:

Record	Count
Tracks	815,631
Tracks With Albums	604,204
Tracks With Tags	363,140
Tracks With Sentiment	361,091
Artists	162,702
Artists With Tags	122,846
Artists With Sentiment	122,646
Albums	384,995
Albums With Tags	127,396
Albums With Sentiment	126,572
Tags	199,608
Tags With Sentiment	177,106
Tag-Track Relationships	2,512,453
Tag-Artist Relationships	980,305
Tag-Album Relationships	733,505
Users	52,829
User-Track Interactions	2,452,162
User-Artist Interactions	527,051
User-Album Interactions	526,855

Table 3.1: Record count of database entities and relationships.

4

Experiments and Results

This chapter exhibits the experiments carried out on the various recommendation models over the collected dataset and analyzes the subsequent testing results. First, the environment used for these tests will be presented, followed by an analysis of the data used for the recommendations and its preparation. Then, the selection process for the recommendation library and models will be explained and, lastly, the results obtained from the models will be laid out and summarized.

4.1. Testing environment

All experiments were conducted on a personal computer, the specifications of which are defined in Table 4.1. It provided an adequate environment for the project's needs by ensuring sufficient memory and processing power to run most of the algorithms efficiently, despite the size of the dataset. This advantage enabled the project to be executed without the additional complexities and costs associated with cloud-based solutions.

Component	Specifications
CPU	AMD Ryzen 5 5600X
GPU	NVIDIA RTX 3060 Ti 8 GB VRAM
RAM	32 GB
O.S.	Windows 11
Python Environment	Python 3.9.16 (conda)

Table 4.1: Specifications of the environment used for experimentation.

The dataset was designed to remain within the hardware constraints of the testing environment, although calculations were not thoroughly performed for all the models. Some of them demanded excessive resources (in the order of 400GB of RAM or more), and had to be excluded from testing. It is worth mentioning that the coding phase of the project was carried out in a different environment, with lower computational capacity and more limited memory. The testing phase was therefore parallelized with the development of the web application.

4.2. Data analysis

The music tags collected for the project amounted to nearly 200,000, and their assignment frequency follows a power-law pattern. In particular, this corresponds to Zipf's law [1], a frequent occurence in text corpus and natural language scenarios, which states that in a given dataset, the frequency of any word or item is inversely proportional to its rank. This means that a few popular tags, like *rock* or *pop*, are assigned to a large number of items, while the majority of tags are much rarer, being associated with only one item, as represented in Figure 4.1. This distribution introduces sparsity in tag frequency, which might adversely impact the recommendation process.

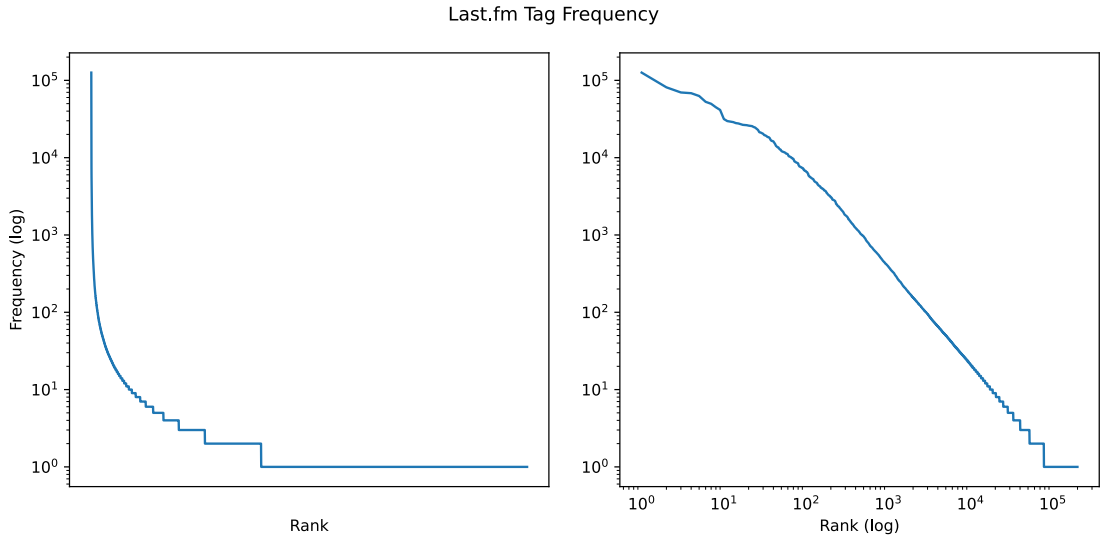


Figure 4.1: Zipf's law in Last.fm's tags

From these tags, VAD scores, along with an additional sentiment ratio, were extracted for approximately 89 % of the tags. Delving into these scores, the correlation heatmap in Figure 4.2 indicates that none of them are negatively correlated. Valence and arousal, being uncorrelated, emerge as promising candidates for embedding into the recommendation models. However, valence and dominance exhibit a higher correlation, suggesting that positive tags (e.g., *happy*, *excited*) tend to be more dominant, and negative tags (e.g., *sad*, *anxious*) are less dominant, with the opposite being less common. Similarly, arousal and dominance also display a high correlation, though still fairly independent.

The sentiment ratio, being dependent on sentiment labels derived from valence, shows a high correlation with valence and dominance, though it is essentially independent of arousal — making it worth considering for inclusion in the models. Therefore, in one combination or another, all of these scores could represent diverse sentiments, which might prove useful for the models as a separate space for searching similarities.

[1] *Zipf's law*, accessed 2023-07-16. https://en.wikipedia.org/wiki/Zipf's_law

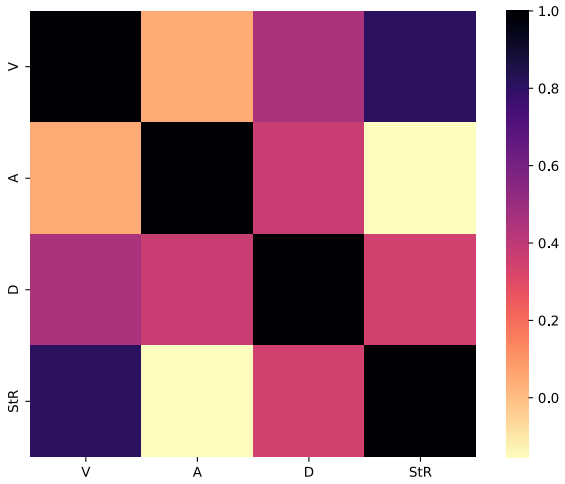
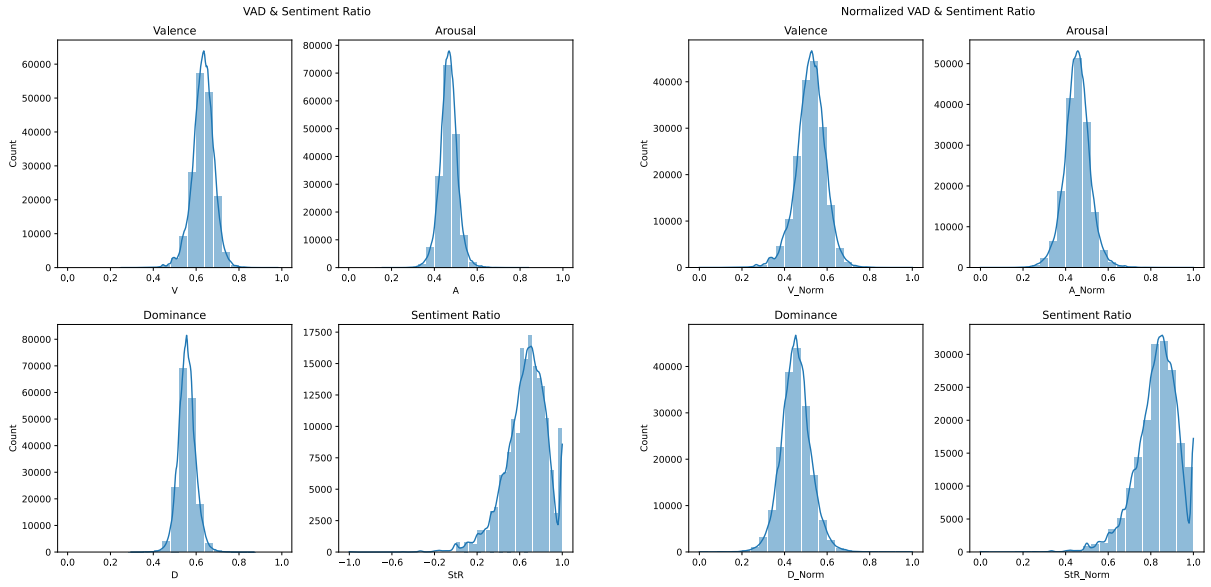


Figure 4.2: Correlation heatmap of VAD & sentiment ratio

On another note, it might be interesting to analyze the distribution of values for these sentiment attributes. The univariate histograms in Subfigure 4.3(a) reveal that most scores fall within the range of $(0{,}4-0{,}8)$ although they should vary between 0 and 1 (or -1 and 1 for sentiment ratio). This narrow clustering of values could potentially hinder recommendation systems, and therefore normalization should be performed to achieve a consistent scale and range, making the features comparable for models to identify patterns in. After normalization between 0 and 1 (Figure 4.3(b)), the values were standardized and expanded, now centered between $0{,}2$ and $0{,}8$; except sentiment ratio, which tended towards the right, indicating that most tags had a higher frequency of positive words associated with them.



(a) Original values

(b) Normalized values

Figure 4.3: Univariate histograms of VAD and sentiment ratio extracted from tags, representing original and normalized values.

Bivariate histograms of the values, as presented in Figure 4.4, further corroborate the correlations previously discussed. The valence-arousal plane exhibits a roughly uniform distribution with a V-shaped pattern, whereas the valence-dominance plane demonstrates a similar distribution but revealing a linear tendency, once again affirming their high correlation. Similarly, the arousal-dominance plane displays a uniform, somewhat more vertical distribution than valence-dominance. The colored spots correspond to the densest areas of values, aligning with the trends observed in the univariate distributions.

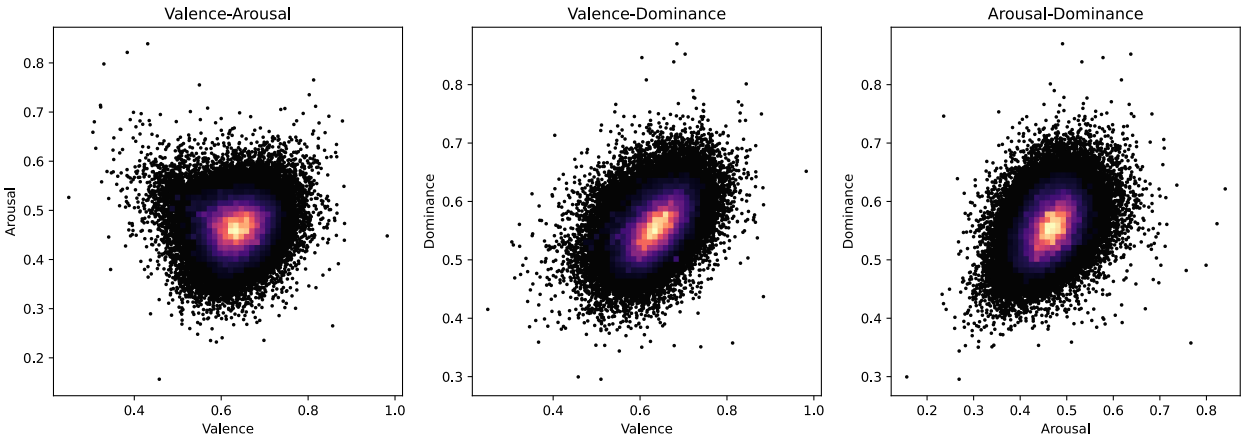


Figure 4.4: Bivariate VAD histograms, each illustrating a sentiment plane.

4.3. Experiments

The experiments were designed to explore various sets of features for recommendation models, thoroughly testing each model with different combinations of features in order to identify the most effective configuration. The training process involved cross-validation with sets for training, validation, and testing. After each training epoch, validation was conducted, and if the validation scores did not improve within a specified number of epochs (mostly 10), the training would conclude; after which the models were evaluated on the testing set. Additionally, negative sampling was employed to help models distinguish relevant items from irrelevant ones.

Two evaluation methods were performed: labeled evaluation for explicit feedback, considering only positive or negative items by a rating threshold, which did not provide enough information to judge the recommendations from; and unlabeled evaluation for implicit feedback, which included ratings and treated seen items as positive. Specifically for the latter, the “uni100” evaluation mode sampled 100 negative items uniformly for each positive item in the testing set and then assessed the model’s performance. Another implicit feedback mode, “full”, was considered, which used the entire dataset for evaluation, but its time-consuming nature made it less practical. Finally, a seed was preset to ensure identical data for all models, and the applied evaluation metrics include NDCG, Recall, Precision, mAP, and MRR, cut off at the 20 first items, with NDCG@20 being the valid metric for early training stopping.

4.3.1. Data used in experiments

Data preprocessing and feature selection

The data utilized for the experiments consisted of all the users, items and interactions between them, along with various item features, imperatively including VAD and sentiment ratio. Prior to incorporating any features, preprocessing steps were carried out, such as assigning ratings to interactions, grouping tags by item, reducing sparsity for tracks' tags and sentiment attributes by using those from artists or albums, and normalization (see Appendix C for a detailed depiction). Several libraries were leveraged during this preparation process, including efficient data processing modules like Numpy [28] and Pandas, as well as visualization toolkits like Matplotlib [29] and seaborn [30]; all integrated into Jupyter Notebooks [31] for interactive data analysis and handling.

For conducting the experiments, different combinations of features were embedded into RecBole models with configuration files, including tokens (artists, albums and tags), and numerical values (sentiment attributes). Moreover, users' top artists and top albums were not included as user features, since they were tested and did not enhance recommendations, presumably being inferred from tracks' artists and albums. Features unrelated to VAD were necessary to provide a fair and rigorous performance comparison, as the results from using only sentiment would not be as insightful.

Feature integration

The process of feature integration was largely automated, since the context-aware recommender models processed them with PyTorch on initialization, as embeddings for neural networks. In the case of *CosineSimilarityRecommender* model, the tags were accessed in the constructor by Scikit-learn [32], to build the vectorizer and create the matrix of vectorized tags, and likewise for *HybridVADRecommender* with sentiment features, to store the necessary variables.

4.3.2. Recommendation libraries and model selection

During the selection process, three main recommendation libraries were considered: Surprise [33], RecBole, and Cornac [34], each presenting distinct advantages:

- **Surprise** stood out for its ease of use and dataset adaptability; however, its limited number of algorithms and lack of documentation for implementing models rendered it inappropriate for this project.
- **RecBole** offered a wide array of algorithms, resourceful documentation and model customization, efficient GPU-accelerated execution with PyTorch and TensorFlow, and data formatting standards. Its use of separate files for data loading, model training and evaluation, allowed for easy configuration.
- **Cornac**, on the other hand, provided many algorithms, but fewer compared to RecBole, with straight-

forward dataset adaptation and model customization as well. Nonetheless, some models faced compatibility issues due to outdated TensorFlow versions.

Ultimately, RecBole emerged as the decisive recommendation library. For model selection within RecBole, preliminary tests were executed by including a simpler set of features, the results of which will be detailed in the next section.

Context-Aware Recommenders exhibited strong performance overall, and were the most fitting for the recommendation task at hand. From these, the three best models, **xDeepFM**, **PNN** and **DCN V2**, would be chosen for evaluation with sentiment features. Some General Recommenders experienced delays in data structures initialization and demanded excessive memory, but most proved effective as well. **Pop** (popularity-based recommendation) and **ItemKNN** (item-based collaborative filtering) were used as baseline models, while **CosineSimilarityRecommender** and **RandomRecommender** were implemented for additional testing.

Sequential Recommenders were not used due to VRAM limitations and the fact that top tracks, the most frequent type, lacked timestamp information needed by them; and Knowledge-Based Recommenders were left out as well because of missing data required to generate the mandatory files. Unfortunately, *HybridVADRecommmender* was not able to improve the scores of the inherited recommenders, as the Euclidean distances between sentiment attributes were probably of little use for such advanced algorithms; and had to be excluded.

4.4. Results

This section studies the results obtained from testing several recommendation models with different combinations of features, starting with an intuitive approach towards sentiment-aware recommendations, then explaining the preliminary results without sentiment features, and concluding with the best combination of this last section, together with VAD and/or sentiment ratio.

Intuition

Sentiment attributes can be easily incorporated into context-aware recommenders, as additional numerical embeddings. As they are relatively independent from each other, there is potential for algorithms, such as Factorization Machines, to infer latent factors from them and uncover additional similarities between tracks, thus enhancing the recommendations. Be that as it may, the true value of these sentiment attributes can only be assessed if they improve the model, even when tags are already included as features. This criterion will demonstrate their independence from tags and their ability to convey additional latent factors or relevant information for the recommendation process.

4.4.1. Performance analysis

Preliminary results

First, the models were tested without any features. In this scenario, the general models performed the best, as they are specialized in working only with user-item interactions; whilst context-aware recommenders, which depend on context to extract similarities from, yielded worse results. From this point on, only context-aware recommenders will be considered, as general recomenders make no use of feature embeddings.

Afterwards, with the inclusion of artists, the accuracy from context-aware recommenders matched those from general recommendation. When albums were then embedded, most models did not improve, and some even worsened, probably due to sparsity as 25 % of tracks lacked an album. After tags were embedded, all recommenders improved by 3 % to 9 % in NDCG, thanks to the role they play as categorizers of tracks, proven efficient for recommendation; also, it was tested that using artists and tags without albums bore better results overall. Therefore, the best setting without sentiment attributes would be using **artists and tags** as embeddings.

Optimal sentiment features

A similar approach was conducted with sentiment features, by testing with several combinations for the purpose of finding the best possible embedding set from valence, arousal, dominance, and sentiment ratio. These tests revealed that using only one attribute as embedding did not lead to significant improvements; however, using two showed better results, the best choice being valence and arousal, which improved some models by nearly 10 % in NDCG, again due probably to both being the most uncorrelated in the VAD spectrum, and therefore showing potential for additional significance.

Interestingly, sentiment ratio and arousal did not contribute much improvement to the models, indicating that they might not effectively represent emotions. On the other hand, using valence, arousal, and sentiment ratio together did lead to some enhancements, suggesting that this combination might allow for a better balance between the subjective and objective aspects of the extracted sentiment.

Following the evaluation of all possible options, the optimal one included all four attributes: VAD and sentiment ratio, although using only VAD was close behind. Notably, the independence of these attributes in some combination or another seemed to be inferable by the models, further substantiating the findings from Section 4.2. From these results, the final feature combination would consist of **artists, tags, valence, arousal, dominance and sentiment ratio**.

Final results

After preliminary testing, only the best context-aware models were evaluated, even though others improved substantially more, for the sake of simplicity. Upon testing with the best possible feature settings, both preliminary and final results are summarized in Table 4.2.

	Model	Testing Results @ 20									
		Preliminary (Artists + Tags)					Final (Artists + Tags + VAD + St.Ratio)				
		NDCG	Recall	Precision	mAP	MRR	NDCG	Recall	Precision	mAP	MRR
General	Random	0.03	0.06	0.01	0.01	0.04	=	=	=	=	=
General	CosineSimilarity	0.14	0.28	0.05	0.06	0.12	=	=	=	=	=
General	Pop	0.31	0.42	0.07	0.20	0.39	=	=	=	=	=
General	ItemKNN [35]	0.44	0.46	0.08	0.33	0.64	=	=	=	=	=
Context	PNN [36]	0.56	0.68	0.12	0.42	0.67	0.58	0.73	0.13	0.44	0.67
Context	xDeepFM [37]	0.57	0.67	0.12	0.43	0.68	0.60	0.76	0.13	0.44	0.67
Context	DCN V2 [38]	0.58	0.71	0.13	0.43	0.66	0.56	0.69	0.12	0.42	0.65

Table 4.2: Preliminary and final testing results.

The general recommenders, as expected, performed worse than the context-aware models. The **Random** model, unsurprisingly ineffective, achieved a Precision of 1 %, which makes sense given that the testing set contains 100 irrelevant items per relevant track, as explained for “uni100”. **Pop** demonstrated good performance, benefiting from a reduced evaluation set that increased the likelihood of common items being included. **CosineSimilarity** also delivered decent results, validating the fact that listeners tend to gravitate towards similar tags. The collaborative filtering model, **ItemKNN**, proved effective in capturing item similarities from user-item interactions, which will always be better than relying only on content, unless the two collaborate in a hybrid model.

In view of the context-aware models, it is worth noting that they are all Deep Learning models (neural networks), allowing to infer functions that effectively adapt to all kinds of problems and data. Moreover, all three algorithms emphasize the importance of feature integration: PNN (Product-based Neural Network) focuses on learning a distributed representation of categorical features, xDeepFM (Deep Factorization Machines) addresses the combination of explicit and implicit feature interactions, and DCN V2 (improved Deep & Cross Network) gives priority to learning efficient feature crosses, which are synthetic features from combining two or more individual features. In addition, they were designed to excel in sparse feature spaces, which matches with the scraped dataset.

Concerning the scores obtained for the context-aware recommenders, both **PNN** and **xDeepFM** managed to enhance the recommendations, particularly for Recall, by 5 % and 9 %, respectively; whereas **DCN V2**’s lowered. This could derive from overfitting, as these models tend to prioritize categorical data when learning at first. Overall and judging from these results, it can be confidently concluded that sentiment attributes do play a role in the representation of people’s taste in music, successfully capturing emotional implications and preference diversity.

5 Conclusions and Future work

5.1. Conclusions

This project has helped shed light on the competence of advanced recommendation algorithms in discerning latent factors and similarities from different track features. Remarkably, the manual assignment of tags by users proved to be the most influential in identifying representative similarities. Therefore, it should come as no surprise that VAD scores are likely contributors to recommendation models, as both tags and sentiment attributes were derived from manual annotations by human beings, with the latter stemming from precise and reliable methodologies. In spite of all the information lost by averaging and using objective text for a task that required subjectivity, relevance was still found in the extracted sentiment. This discovery emphasizes the dual significance of tags, providing a logical search space, and their sentiment attributes, offering an emotional dimension which, as evidenced in this work, was indeed significant in shaping musical preferences.

Delving into sentiment analysis and NLP, despite the initial challenges, turned out to be highly captivating, as addressing the intricacies of sentiment attributes in music demanded innovative problem-solving; from the point of view of an undergraduate student, that is. The vast potential of sentiment analysis evoked an interest to further explore and contribute to its ongoing development and application in diverse fields.

In summary, this project served as a testament to the acquired knowledge and skills in web development, software engineering, and machine learning over the years; as well as a platform for exploring uncharted territories, namely NLP, sentiment analysis, web scraping, and music recommendation, thus broadening the horizons of expertise. Ultimately, this Bachelor Thesis has not only contributed to the understanding of new and novel topics, but has also enriched the journey of intellectual growth and personal development.

5.2. Future work

With reference to the topics discussed thoughout this project, several aspects may be addressed hereafter. To begin with, all the tested recommendation models could be enhanced with exhaustive hyperparameter tuning, indispensable to achieve optimal results; and newer models or libraries could be explored to diversify the scope and types of recommendation algorithms.

To obtain real feedback, testing with external, real-world users could also be conducted, both on the performance and accuracy of the models, as well as the user experience of the developed web application.

Regarding the scraped dataset, a suggestion would be making it dynamic by allowing the addition of new users, which entails retraining the models. For this, the project might delve into the use of distributed or parallel computing with technologies like Spark, Hadoop, containerized environments or cloud services for efficient model deployment.

To evaluate the sentiment analyzer, formal accuracy testing would be an option to assess its real efficacy, and alternative sources for extracting sentiment attributes, such as lyrics from tracks, could be considered to achieve more representative and subjective text for sentiment analysis.

Last but not least, future research shall focus on broadening the understanding of VAD or different sentiment models, aiming to refine sentiment analysis in music recommendation or other domains, and potentially uncovering novel insights for the coming years.

Bibliography

[1] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender Systems Handbook*. Springer, 2022.

[2] Y. Koren, S. Rendle, and R. Bell, *Advances in Collaborative Filtering*, pp. 91–142. In [1], 2022.

[3] G. Adomavicius, K. Bauman, A. Tuzhilin, and M. Unger, *Context-Aware Recommender Systems: From Foundations to Recent Developments*, pp. 211–250. In [1], 2022.

[4] M. Schedl, P. Knees, B. McFee, and D. Bogdanov, *Music Recommendation Systems: Techniques, Use Cases, and Challenges*, pp. 927–971. In [1], 2022.

[5] C. Musto, M. de Gemmis, P. Lops, F. Narducci, and G. Semeraro, *Semantics and Content-Based Recommendations*, pp. 251–298. In [1], 2022.

[6] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, "spaCy: Industrial-strength Natural Language Processing in Python," 2020.

[7] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc., 2009.

[8] Wikipedia, "Emotion classification — Wikipedia, The Free Encyclopedia," 2023.

[9] W. Commons, "Circumplex model of emotion," 2023.

[10] W. Commons, "PAD emotional state model," 2023.

[11] D. T. Rubin and J. M. Talarico, "A comparison of dimensional models of emotion: Evidence from emotions, prototypical events, autobiographical memories, and words," *Memory*, vol. 17, pp. 802–808, 11 2009.

[12] S. Wu, "Design your own Sentiment Score," *Towards Data Science*, 5 2021.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[14] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1532–1543, Association for Computational Linguistics, Oct. 2014.

[15] S. M. Mohammad, "Obtaining reliable human ratings of valence, arousal, and dominance for 20,000 english words," in *Proceedings of The Annual Conference of the Association for Computational Linguistics (ACL)*, (Melbourne, Australia), 2018.

[16] A. Ihtsham, "The Top Web Development Frameworks of 2023," *www.linkedin.com*, 3 2023.

[17] Django Software Foundation, "Django."

[18] M. Bayer, "SQLAlchemy," in *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks* (A. Brown and G. Wilson, eds.), aosabook.org, 2012.

[19] W. X. Zhao, S. Mu, Y. Hou, Z. Lin, Y. Chen, X. Pan, K. Li, Y. Lu, H. Wang, C. Tian, Y. Min, Z. Feng, X. Fan, X. Chen, P. Wang, W. Ji, Y. Li, X. Wang, and J.-R. Wen, "Recbole: Towards a unified,

comprehensive and efficient framework for recommendation algorithms," 2021.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[21] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[22] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, vol. 445, pp. 51–56, Austin, TX, 2010.

[23] A. Hassan and Contributors, "pylast." `https://github.com/pylast/pylast`, 2009.

[24] J. Goldsmith and Wikimedia Foundation, "Wikipedia." `https://github.com/goldsmith/Wikipedia`, 2013.

[25] D. Zhou, "SentimentAnalysis." `https://github.com/dwzhou/SentimentAnalysis`, 2017.

[26] C. Hutto and E. Gilbert, "VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text," 01 2015.

[27] P.-L. Glaser, G. Hammerschmied, H. Vladyslav, C. Lauscher, and D. Bork, "bigER." `https://github.com/borkdominik/bigER`, 2021.

[28] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[29] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[30] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021.

[31] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (F. Loizides and B. Schmidt, eds.), pp. 87 – 90, IOS Press, 2016.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and

E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[33] N. Hug, “Surprise: A python library for recommender systems,” *Journal of Open Source Software*, vol. 5, no. 52, p. 2174, 2020.

[34] A. Salah, Q.-T. Truong, and H. W. Lauw, “Cornac: A comparative framework for multimodal recommender systems,” *Journal of Machine Learning Research*, vol. 21, no. 95, pp. 1–5, 2020.

[35] M. Deshpande and G. Karypis, “Item-based top-n recommendation algorithms,” *ACM Trans. Inf. Syst.*, vol. 22, p. 143–177, jan 2004.

[36] Y. Qu, H. Cai, K. Ren, W. Zhang, Y. Yu, Y. Wen, and J. Wang, “Product-based neural networks for user response prediction,” in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 1149–1154, Dec 2016.

[37] J. Lian, X. Zhou, F. Zhang, Z. Chen, X. Xie, and G. Sun, “Xdeepfm: Combining explicit and implicit feature interactions for recommender systems,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD ’18, (New York, NY, USA), p. 1754–1763, Association for Computing Machinery, 2018.

[38] R. Wang, R. Shivanna, D. Cheng, S. Jain, D. Lin, L. Hong, and E. Chi, “Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems,” in *Proceedings of the Web Conference 2021*, WWW ’21, (New York, NY, USA), p. 1785–1797, Association for Computing Machinery, 2021.

Acronyms

AJAX Asynchronous JavaScript And XML.

AP Average Precision.

API Application Programming Interface.

CPU Central Processing Unit.

CSS Cascading Style Sheets.

DCG Discounted Cumulative Gain.

DL Deep Learning.

FM Factorization Machines.

GPU Graphics Processing Unit.

HTML HyperText Markup Language.

IDCG Ideal Discounted Cumulative Gain.

IDF Inverse Document Frequency.

IR Information Retrieval.

JS JavaScript.

mAP Mean Average Precision.

MRR Mean Reciprocal Rank.

MVC Model-View-Controller.

MVT Model-View-Template.

NDCG Normalized Discounted Cumulative Gain.

NLP Natural Language Processing.

NLTK Natural Language Toolkit.

NRC National Research Council.

ORM Object-Relational Mapping.

POS Part-of-speech.

RR Reciprocal Rank.

SQL Structured Query Language.

TF Term Frequency.

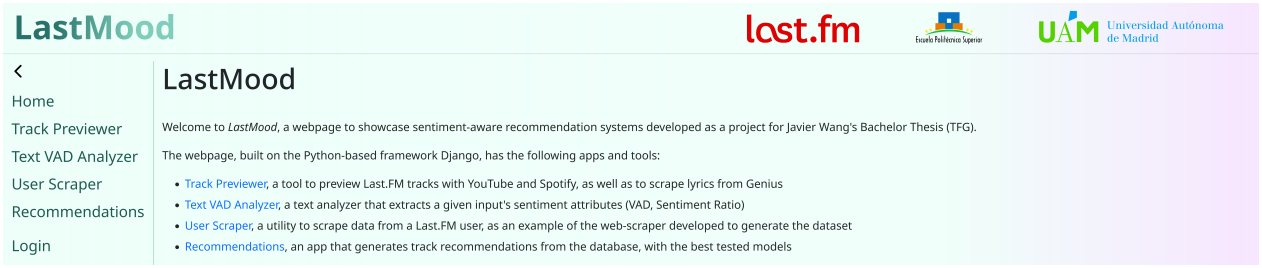
URL Uniform Resource Locator.

VAD Valence-Arousal-Dominance.

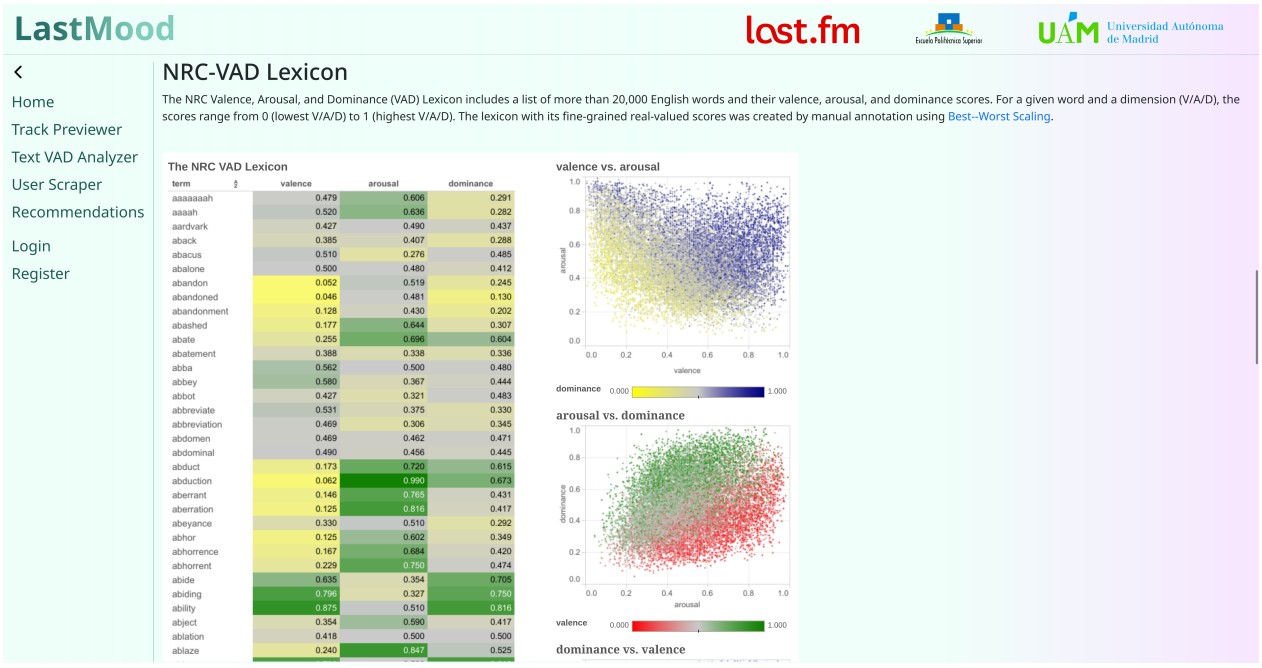
APPENDICES

A

Web Application

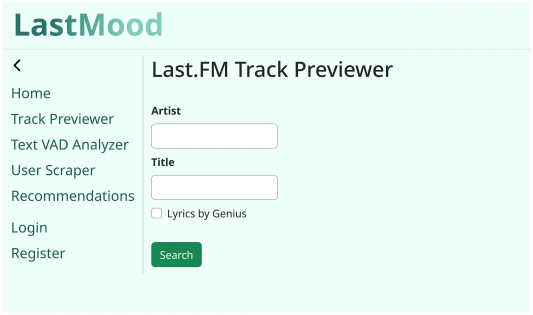


(a) Tools introduction

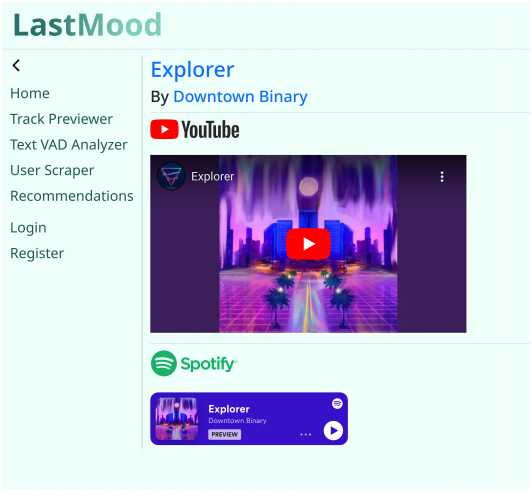


(b) Interactive NRC-VAD lexicon embed

Figure A.1: Web application: Home Page

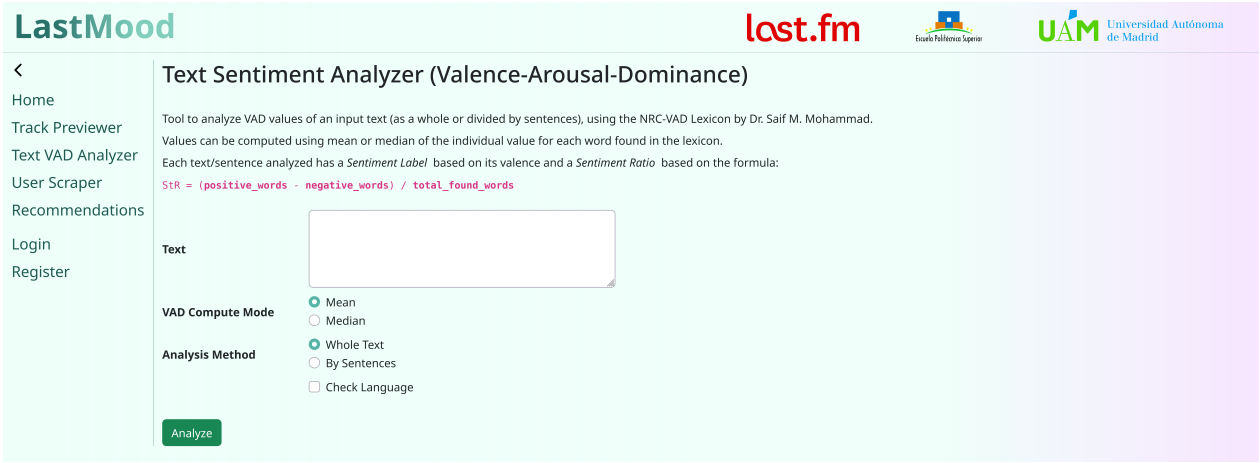


(a) Track previewer form

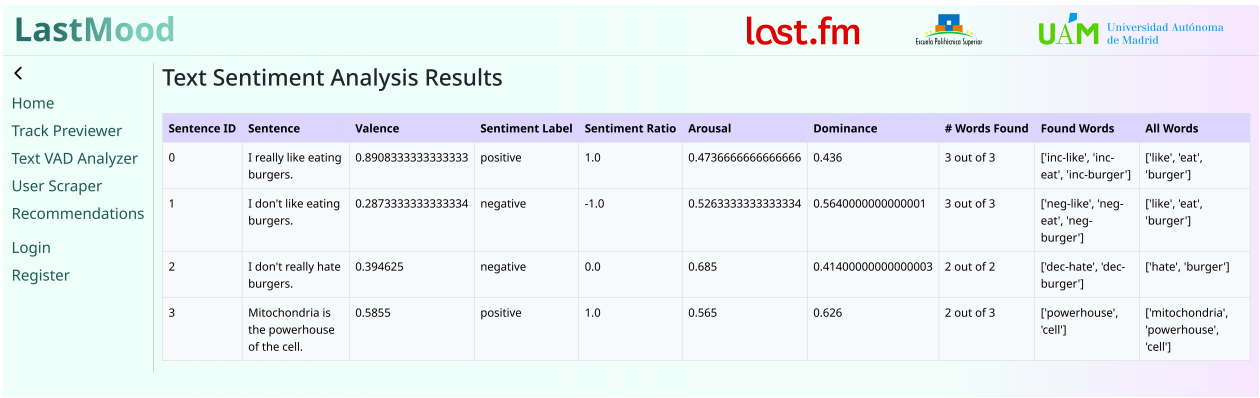


(b) Example preview with artist *Downtown Binary* and title *Explorer*

Figure A.2: Web application: Track Previewer



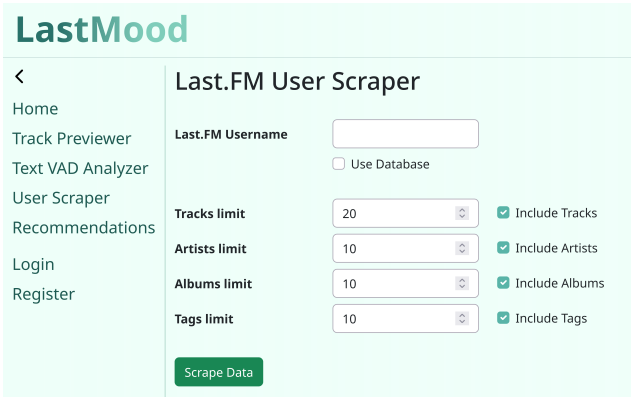
(a) VAD analyzer form



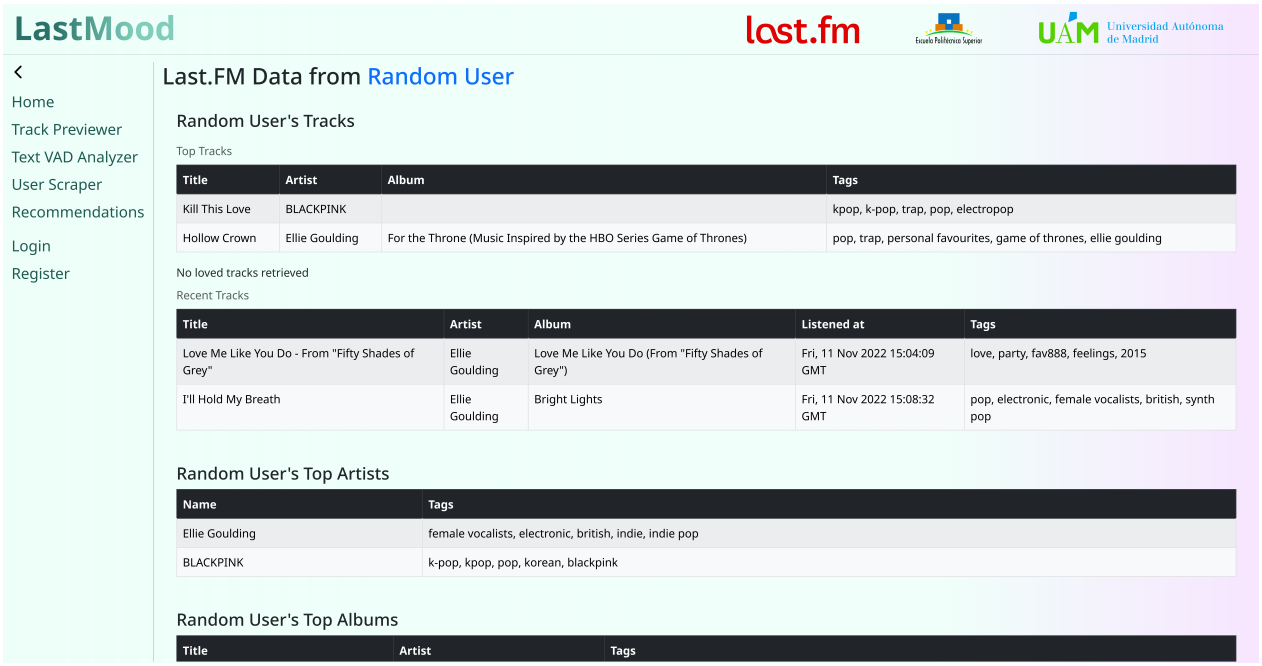
Sentence ID	Sentence	Valence	Sentiment Label	Sentiment Ratio	Arousal	Dominance	# Words Found	Found Words	All Words
0	I really like eating burgers.	0.8908333333333333	positive	1.0	0.4736666666666666	0.436	3 out of 3	['inc-like', 'inc-eat', 'inc-burger']	['like', 'eat', 'burger']
1	I don't like eating burgers.	0.2873333333333334	negative	-1.0	0.5263333333333334	0.5640000000000001	3 out of 3	['neg-like', 'neg-eat', 'neg-burger']	['like', 'eat', 'burger']
2	I don't really hate burgers.	0.394625	negative	0.0	0.685	0.41400000000000003	2 out of 2	['dec-hate', 'dec-burger']	['hate', 'burger']
3	Mitochondria is the powerhouse of the cell.	0.5855	positive	1.0	0.565	0.626	2 out of 3	['powerhouse', 'cell']	['mitochondria', 'powerhouse', 'cell']

(b) Example analysis by sentences

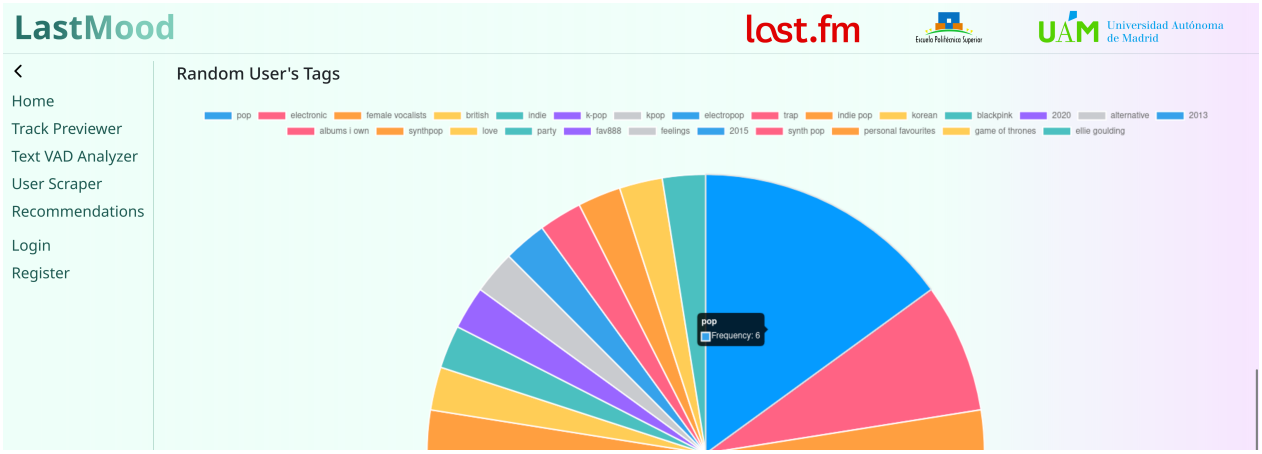
Figure A.3: Web application: VAD Analyzer



(a) User scraper form

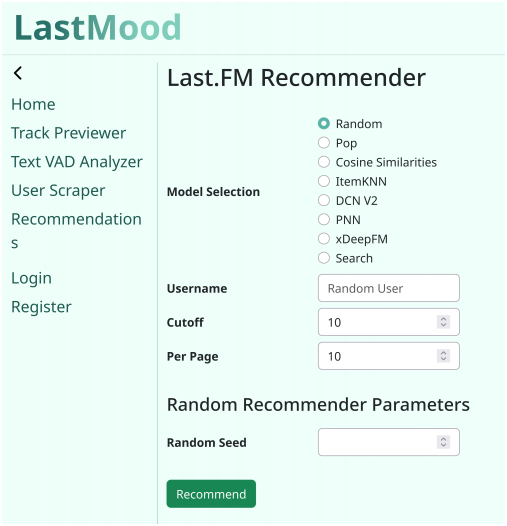


(b) Example with limited results

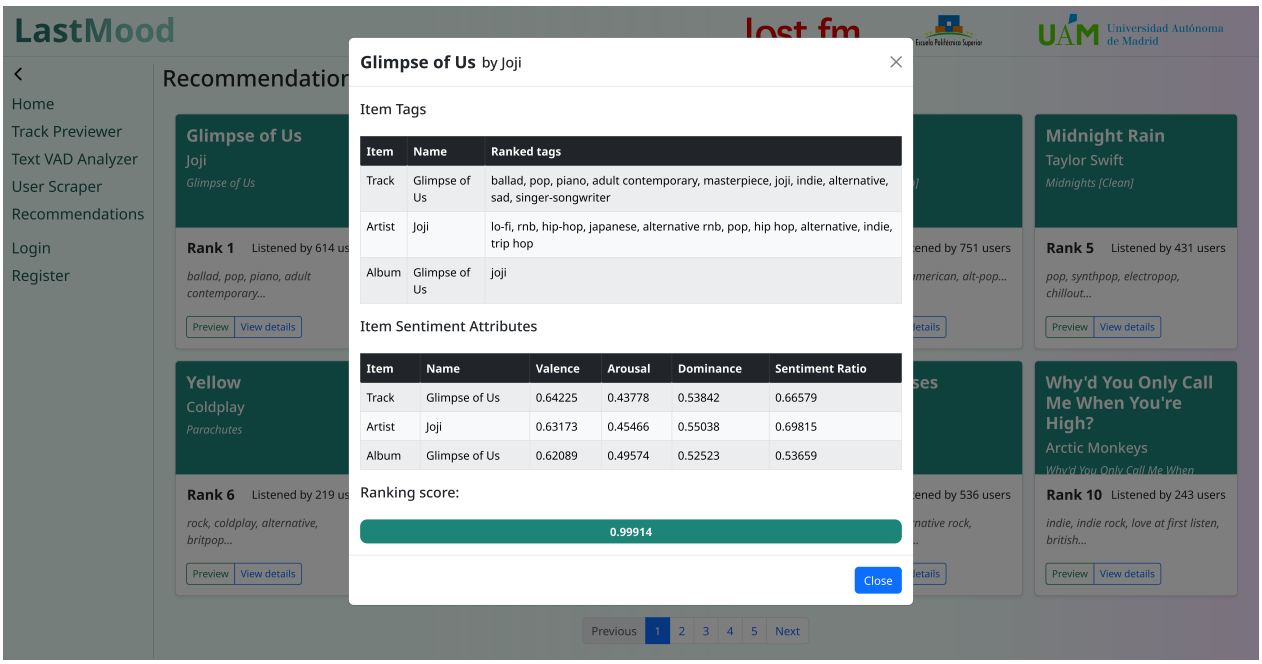


(c) Tag frequency pie chart

Figure A.4: Web application: User Scraper



(a) Track recommender form



(b) Track details modal

Figure A.5: Web application: Recommendations

B

External Libraries

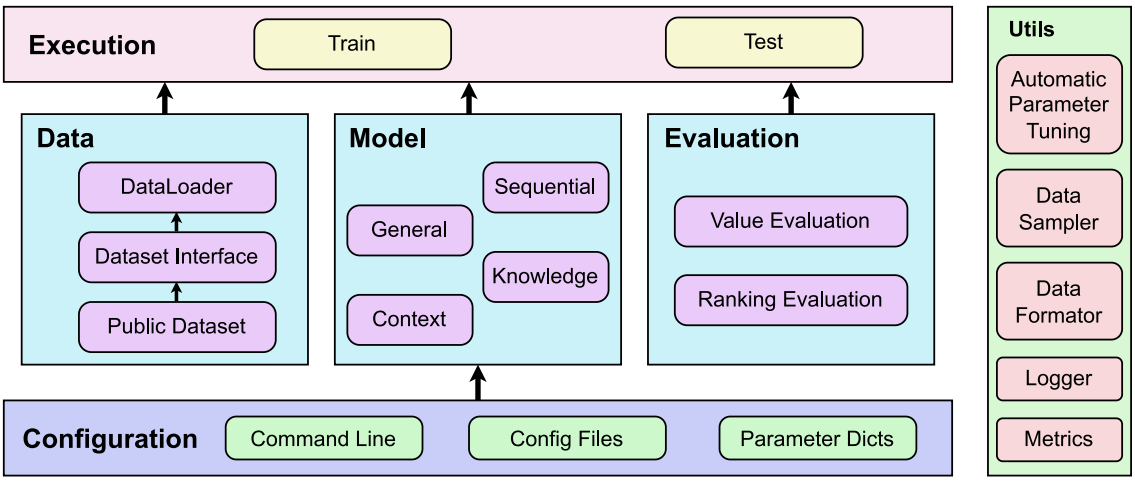


Figure B.1: RecBole Framework [19]

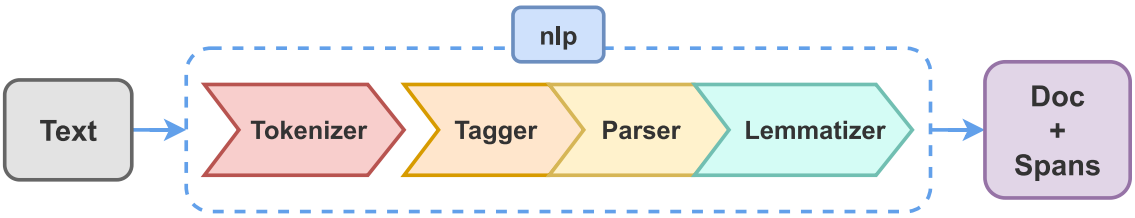


Figure B.2: spaCy NLP Pipeline

C

Data Preprocessing

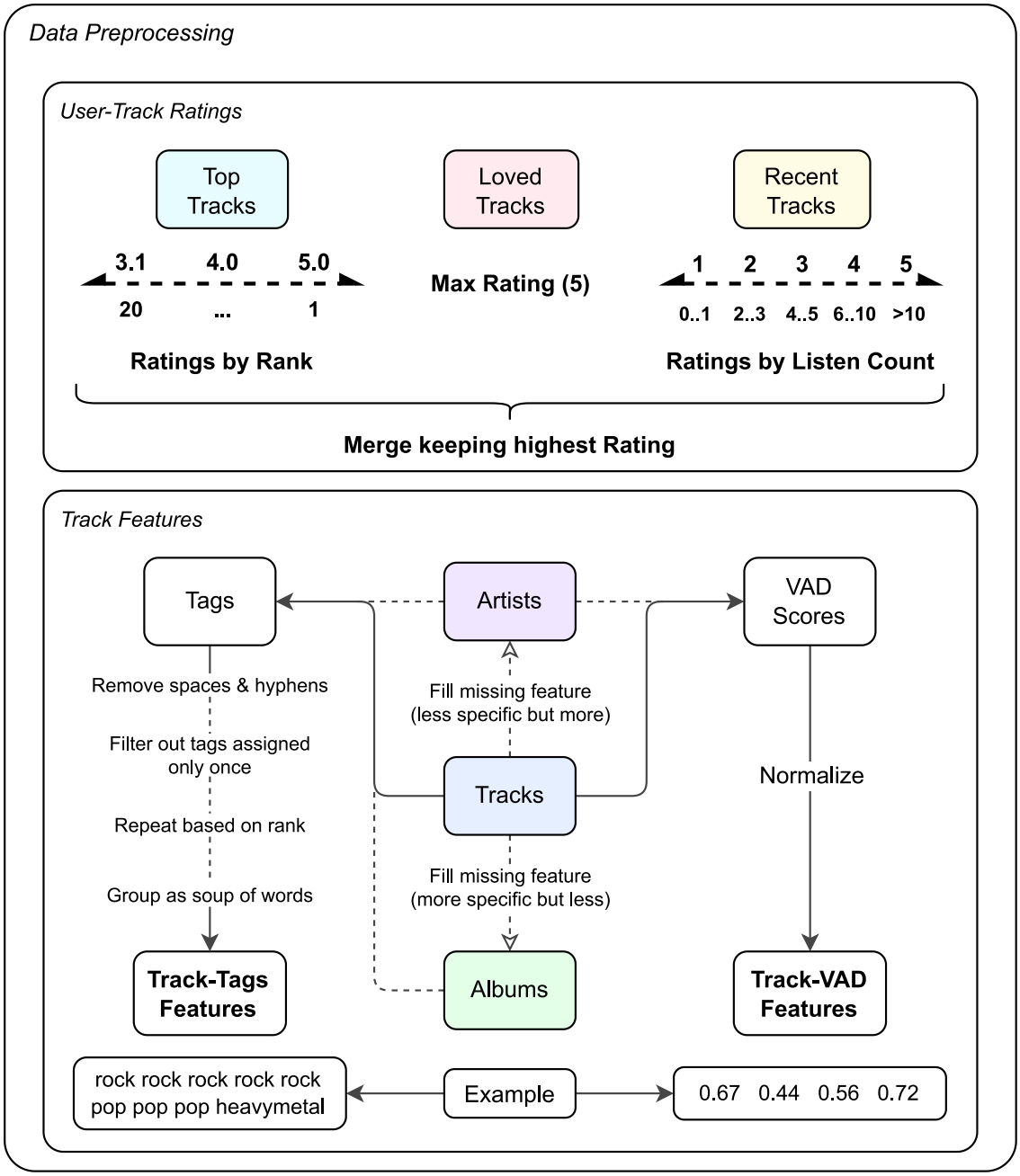


Figure C.1: Data Preprocessing

D

Model Implementations

Code D.1: *RandomRecommender* class for random recommendations as baseline.

```
class RandomRecommender(GeneralRecommender):
    """ Random recommendations as baseline for performance comparison """
    rand_scores: torch.Tensor
    input_type = InputType.POINTWISE
    type = ModelType.TRADITIONAL

    def __init__(self, config, dataset: Dataset):
        super(RandomRecommender, self).__init__(config, dataset)
        # Initial random scores
        self.generate_random_scores()
        # Fake loss parameter
        self.fake_loss = torch.nn.Parameter(torch.zeros(1))

    def calculate_loss(self, interaction):
        # Re-randomize scores
        self.generate_random_scores()
        return torch.nn.Parameter(torch.zeros(1))

    def predict(self, interaction, generate_new=False):
        if generate_new:
            self.generate_random_scores()
        item = interaction[self.ITEM_ID]
        return self.rand_scores[item, :].squeeze(-1)

    def full_sort_predict(self, interaction, generate_new=False):
        if generate_new:
            self.generate_random_scores()
        batch_user_num = interaction[self.USER_ID].shape[0]
        result = torch.repeat_interleave(self.rand_scores.unsqueeze(0), batch_user_num, dim=0)
        return result.view(-1)

    def generate_random_scores(self):
        # Generate random tensor of scores
        self.rand_scores = torch.rand(self.n_items, 1, device=self.device)
```

Code D.2: *CosineSimilarityRecommender* class for tag similarity recommendations.

```
class CosineSimilarityRecommender(GeneralRecommender):
    """ Recommendations based on features' (tags) cosine similarities """
    input_type = InputType.PAIRWISE
    type = ModelType.TRADITIONAL

    def __init__(self, config, dataset: Dataset):
        super(CosineSimilarityRecommender, self).__init__(config, dataset)
        # Model config, tag features, item indexing, similarity weights, topk cutoff, fake loss...
        ...
        # Set up vectorizer
        self.vectorizer = TfidfVectorizer(**config['Vectorizer_Config'], token_pattern=r"(?u)\b\w+\b",
            lowercase=False)
        # Vectorized item-tags matrix, vec_feat contains all the tag sequences
        self.vec_matrix = normalize(self.vectorizer.fit_transform(self.vec_feat))

    def cosine_similarity_scores(self, user_id):
        # User interactions indices
        users = self.inters[self.USER_ID]
        user_inters_idx = users == user_id.item()
        # Interacted items
        user_items = self.inters[self.ITEM_ID][user_inters_idx]
        # Item weights by rating
        weights = None
        if self.sim_weights is not None:
            weights = self.sim_weights[user_inters_idx]
            # Sort by weights for cutoff
            idx_by_weights = np.argsort(-weights)[:self.knn_topk]
            weights = weights[idx_by_weights]
            user_items = user_items[idx_by_weights]
        # Features to vectorize for user items
        items_idx = self.item_to_idx.loc[user_items.flatten()]
        rec_items_feat = self.vec_feat[items_idx]
        return self.feature_cosine_scores(rec_items_feat, items_idx, weights)

    def feature_cosine_scores(self, rec_items_feature, items_idx=None, item_weights=None):
        # Vectorize selected features
        rec_matrix = self.vectorizer.transform(rec_items_feature)
        # Compute cosine similarities with all items
        rec_matrix_norm = normalize(rec_matrix, copy=True)
        sims = safe_sparse_dot(self.vec_matrix, rec_matrix_norm.T, dense_output=True)
        # Cancel items used for recommendation
        if items_idx is not None:
            sims[items_idx] = 0
        # Average feature similarities
        return np.average(sims, weights=item_weights, axis=1)

    def calculate_loss(self, interaction): ...
    def predict(self, interaction): ...
    def full_sort_predict(self, interaction): ...
```

Code D.3: *HybridVADRecommender* class for hybrid recommendations, computing the Euclidean distance of VAD scores and averaging with other models.

```python
class HybridVADRecommender(ContextAwareModel):
    """ Recommendations from averaging VAD distances with scores from other models """

    def __init__(self, config, dataset: Dataset):
        super().__init__(config, dataset)
        # Model config, VAD features, item indexing, similarity weights, topk cutoff, fake loss...
        ...
        # VAD centroids for all users
        self.sentiment_centroids = self.compute_sentiment_centroids()

    def compute_sentiment_centroids(self):
        # vadst contains VAD scores of every item
        sentiment_centroids = [np.array([0] *self.vadst.shape[1])]
        for user in tqdm.trange(1, self.n_users, desc='Computing user centroids'):
            user_inters_idx = self.inters[self.USER_ID] == int(user)
            # Interacted items
            user_items = self.inters[self.ITEM_ID][user_inters_idx]
            # Item weights by rating
            weights = None
            if self.sim_weights is not None:
                weights = self.sim_weights[user_inters_idx]
                # Sort by weights for cutoff
                idx_by_weights = np.argsort(-weights)[:self.knn_topk]
                weights = weights[idx_by_weights]
                user_items = user_items[idx_by_weights]
            # Average items VAD
            items_idx = self.item_to_idx.loc[user_items.flatten()]
            user_vads_centroid = np.average(self.vadst[items_idx], weights=weights, axis=0)
            sentiment_centroids.append(user_vads_centroid)
        return np.array(sentiment_centroids)

    def sentiment_knn_scores(self, user_id, item_id=None):
        user_centroid = self.sentiment_centroids[user_id.item()]
        # VADSt euclidean distances
        item_vadst = self.vadst if item_id is None else self.vadst[item_id.cpu().numpy()]
        distances = cdist(item_vadst, user_centroid.reshape(1, -1), 'seuclidean')
        # Inverted average euclidean distances as scores
        return 1 / (1 + distances)

    def hybrid_scores(self, model_scores, scores):
        model_scores = normalize(model_scores.cpu().detach().numpy().reshape(1, -1))
        scores = normalize(scores.reshape(1, -1))
        # Score similarities between both models
        sims = 1 / (1 + np.abs(scores -model_scores))
        # Add scores buffing similar ones
        return model_scores + (sims *scores)

    def predict(self, interaction): ...
    def full_sort_predict(self, interaction): ...
```


UÁM
Universidad Autónoma
de Madrid