PGCOMP - Programa de Pós-Graduação em Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Milton Santos, s/n - Ondina
Salvador, BA, Brasil, 40170-110

https://pgcomp.ufba.br
pgcomp@ufba.br

In the rapidly evolving digital landscape, users are often overwhelmed by the multitude of listing services, ranging from music platforms to product recommenders and social media content suggestions, leading to a challenge in finding items that align with their individual preferences. To address this complexity, the development and implementation of Recommender Systems has become increasingly valuable. These systems efficiently sift through large volumes of data to match items with user preferences, thereby enhancing the choices available to users. The focus of this work is on the development of an advanced Application Programming Interface (API) for Recommender Systems. This Application Programming Interface is uniquely designed to be universally accessible and easy to deploy. Serving as the backbone for various Web Services, the Application Programming Interface utilizes the robust Representational State Transfer architecture. It is crafted with an emphasis on modularity, promoting adaptability and flexibility. The Application Programming Interface processes user data and queries to deliver customized recommendations swiftly. Performance evaluations have demonstrated the commendable accuracy of the Application Programming Interface. It exhibits outstanding performance particularly with smaller datasets, showcasing rapid data processing and algorithm execution times. The Application Programming Interface has shown exceptional efficiency and resilience under specific testing conditions, including cloud environments, and this is particularly evident in scenarios involving extensive datasets of up to 16,000 items. The Application Programming Interface is more than a mere tool; it represents a pathway towards personalized digital experiences, excelling in Create, Read, Update, and Delete operations and customized recommendations. The user evaluation phase included a diverse group of participants, ranging from novice to experienced developers. Over half of these participants had substantial experience in software development, and a significant proportion had previously worked with coding recommender systems. Given their varied knowledge of recommender libraries, most feedback commended the effectiveness of the Application Programming Interface. 81% of users valued the recommendations provided, and many expressed confidence in its filtering techniques. The standout feature of this work is the versatility of the Recommender System Application Programming Interface. Despite the positive feedback, users suggested improvements in areas such as documentation, data security, and features. These insights are valuable for future refinements of the Application Programming Interface and the user experience. The enthusiastic engagement and feedback from participants underscore the potential of the Application Programming Interface to enhance applications that require a recommendation system, particularly for developers who may not be as familiar with the theoretical aspects. The solid research foundation and the dedication of the participants highlight the potential for broader adoption of the Application Programming Interface by developers.

Keywords: Software as a Service, API, Recommender Systems, RecSys, Microservices, Cloud.

MSC | 169 | 2023

An intelligent self-configuring Recommender System as a Service

Felipe Rebouças Ferreira Abreu

UFBA

An intelligent self-configuring Recommender System as a Service

Felipe Rebouças Ferreira Abreu

Dissertação de Mestrado

Universidade Federal da Bahia

Programa de Pós-Graduação em
Ciência da Computação

November | 2023

Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

AN INTELLIGENT SELF-CONFIGURING RECOMMENDER SYSTEM AS A SERVICE

Felipe Rebouças Ferreira Abreu

DISSERTAÇÃO DE MESTRADO

Salvador
Novembro de 2023

FELIPE REBOUÇAS FERREIRA ABREU

AN INTELLIGENT SELF-CONFIGURING RECOMMENDER SYSTEM AS A SERVICE

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Computer Science.

Advisor: Frederico Araújo Durão

Salvador
Novembro de 2023

Ficha catalográfica elaborada pela Biblioteca Universitária de
Ciências e Tecnologias Prof. Omar Catunda, SIBI – UFBA.

A162 Abreu, Felipe Rebouças Ferreira

An intelligent self-configuring recommender system as a service/ Felipe Rebouças Ferreira Abreu. – Salvador, 2023.

93 f.

Orientador: Prof. Dr. Frederico Araújo Durão

Dissertação (Mestrado) – Universidade Federal da Bahia, Instituto de Computação, 2023.

1. Recommender Systems. 2. Cloud. 3. API. 4. Computação. I. Durão, Frederico Araújo. II. Universidade Federal da Bahia. III. Título.

CDU 004

TERMO DE APROVAÇÃO

FELIPE REBOUÇAS FERREIRA ABREU

AN INTELLIGENT SELF-CONFIGURING RECOMMENDER SYSTEM AS A SERVICE

Esta Dissertação de Mestrado foi julgada adequada à obtenção do título de Mestre em Computer Science e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia.

Salvador, 28 de Novembro de 2023

Frederico Araujo Durão

Prof. Dr.Frederico Araujo Durão (Orientador - PGCOMP)

[illegible]

Prof. Dr.Rosalvo Ferreira De Oliveira Neto (UNIVASF)

[illegible]

Prof. Dr. Cláudio Nogueira Sant'Anna (UFBA)

Dedico este trabalho aos meus familiares pelo apoio durante toda esta jornada, ao meu Orientador, Frederico Duro, e a toda a comunidade Acadmica UFBA pela excelncia e qualidade de ensino.

SUMMARY

In the rapidly evolving digital landscape, users are often overwhelmed by the multitude of listing services, ranging from music platforms to product recommenders and social media content suggestions, leading to a challenge in finding items that align with their individual preferences. To address this complexity, the development and implementation of Recommender Systems has become increasingly valuable. These systems efficiently sift through large volumes of data to match items with user preferences, thereby enhancing the choices available to users. The focus of this work is on the development of an advanced Application Programming Interface (API) for Recommender Systems. This Application Programming Interface is uniquely designed to be universally accessible and easy to deploy. Serving as the backbone for various Web Services, the Application Programming Interface utilizes the robust Representational State Transfer architecture. It is crafted with an emphasis on modularity, promoting adaptability and flexibility. The Application Programming Interface processes user data and queries to deliver customized recommendations swiftly. Performance evaluations have demonstrated the commendable accuracy of the Application Programming Interface. It exhibits outstanding performance particularly with smaller datasets, showcasing rapid data processing and algorithm execution times. The Application Programming Interface has shown exceptional efficiency and resilience under specific testing conditions, including cloud environments, and this is particularly evident in scenarios involving extensive datasets of up to 16,000 items. The Application Programming Interface is more than a mere tool; it represents a pathway towards personalized digital experiences, excelling in Create, Read, Update, and Delete operations and customized recommendations. The user evaluation phase included a diverse group of participants, ranging from novice to experienced developers. Over half of these participants had substantial experience in software development, and a significant proportion had previously worked with coding recommender systems. Given their varied knowledge of recommender libraries, most feedback commended the effectiveness of the Application Programming Interface. 81% of users valued the recommendations provided, and many expressed confidence in its filtering techniques. The standout feature of this work is the versatility of the Recommender System Application Programming Interface. Despite the positive feedback, users suggested improvements in areas such as documentation, data security, and features. These insights are valuable for future refinements of the Application Programming Interface and the user experience. The enthusiastic engagement and feedback from participants underscore the potential of the Application Programming Interface to enhance applications that require a recommendation system, particularly for developers who may not be as familiar with the theoretical aspects. The solid research foundation and the dedication of the participants highlight the potential for broader adoption of the Application Programming Interface by developers.

Keywords: Software as a Service, API, Recommender Systems, RecSys, Microservices, Cloud.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CB	Content-Based Filtering	62
CF	Collaborative Filtering	xiii
JSON	JavaScript Object Notation	23
RDMS	Relational Database Management System	35
SaaS	Software as a Service	27
IaaS	Infrastructure as a Service	27
PaaS	Platform as a Service	27
SVD	Singular Value Decomposition	16
KNN	K - Nearest Neighbors	15
HTML	HyperText Markup Language	23
HTTP	Hyper Text Transfer Protocol	23
HTTPS	Hyper Text Transfer Protocol Secure	23
TCP	Transmission Control Protocol	23
CSS	Cascading Style Sheets	23
IP	Internet Protocol	23
URL	Uniform Resource Locator	10
SOA	Service-Oriented Architecture	23

Chapter 1

INTRODUCTION

With the rapid increase of information available through online services, several new types of services have emerged (e.g., music, movies, news, travel guides, etc.) that can list an overwhelming amount of information to the user with a large number of options. With such a large selection, finding an offer that meets the user's preferences requires a lot of effort and attention from the user, who in turn may have difficulty finding offers that he or she does not like. For this user, the large selection is therefore more of a disadvantage than an advantage (RICCI; ROKACH; SHAPIRA, 2011a; HIJIKATA; IWAHAMA, 2006).

Recommender Systems emerge with the goal of filtering relevant information, presenting a ranked list of items based on their preferences, and thus enabling decision making. According to Ricci, Rokach e Shapira (2011a), "Recommender Systems (RSs) are software tools and techniques providing suggestions for items to be of use to a user". Fields et al. (2011) defines Recommender Systems as "a technique or method that presents a user with suggested objects for consumption based on past behaviour". Collaborative filtering and content-based filtering (also known as the personality-based approach) are techniques commonly used in recommender systems, as are other systems such as knowledge-based systems (JAFARKARIMI; SIM; SAADATDOOST, 2012). Collaborative filtering techniques create a model based on a user's previous behavior (things purchased or picked in the past and/or numerical ratings provided to those items) as well as comparable decisions made by other users. This model is then used to estimate which things (or item ratings) the user would be interested in (MELVILLE; SINDHWANI, 2010). Content-based filtering methods use a set of distinct, pre-tagged features of an item to suggest other items with comparable characteristics (MOONEY; ROY, 1999).

There are a number of recommendation services available online, including playlist generators for video and music services, product recommenders for online businesses, content recommenders for social media platforms, and open web content recommenders are all examples of recommender systems in use (GUPTA et al., 2013; BARAN; DZIECH; ZEJA, 2018). These systems can work with a single input, such as music, or numerous inputs, such as news, books, and search queries, within and across platforms. Popular recommender systems for specialized themes such as restaurants and online dating are also available. Research papers, experts, collaborators, and financial services have all been explored using recommender systems as well (FELFERNIG et al., 2007; CHEN; II; GILES, 2015; CHEN et al., 2011).

Academically, Recommender Systems have been the subject of research in a variety of areas, including: explanation of how the recommendations were generated (MITTELSTADT; RUSSELL; WACHTER, 2019); new user and/or item in the system, a problem known as Cold-Start (BARJASTEH et al., 2015); use of linked and open data on the internet to increase information about items (PESKA; VOJTAS, 2013); diversity of items that make up the recommendations (CHENG et al., 2017); popular items and popularity bias (ABDOLLAHPOURI; BURKE; MOBASHER, 2017); fairness in the recommendations (STECK, 2018); among other works.

In general, implementations of recommender systems are available through software libraries that need to be installed, understood by the developer (through documentation), and only then implemented, thus there is complexity. For these reasons, implementing a recommender system in an application requires a high level of technical knowledge (VINEELA et al., 2021). This work envisions an implementation of a Recommender System solution within a software, believing it to be better provided in a modular and loosely coupled way, where the service is agonistic to the current established system architecture, server, frameworks, libraries and programming languages. This way, a modular recommender system, provided as a separate service, can potentially provide an optimal solution to software, platforms and services making use of Service-oriented Architecture, modularity, and abandoning an aging monolithic approach and providing more choices to the developers, as the recommendation functionality will be available to them regardless of their software design choices or programming languages and environment.

1.1 MOTIVATION

Despite of being crucial to e-commerce and media services platforms, Recommender Systems implementations are still far from simplicity (MACMANUS, 2009). Regular web-based system developers might be required to learn about algorithmic structure beyond their knowledge. In this case, an average system designer would still need additional background on machine learning, user behavior modeling, and social network analytics (FAYYAZ et al., 2020). Due to these peculiarities, finding developers with this particular background is an obstacle for those seeking to incorporate recommender systems into their business. Moreover, Web developers can be overloaded with such tasks as implementing recommender systems. Additionally, machine learning libraries (and recommender system libraries in particular) are required to be installed on the system and maintained by the developers, and considering how much of a challenge it can become both due technological choices and time constraints, providing a recommendation API (using known industrial standards such as the REST architecture), which is easily deployable or readily available could significantly reduce the amount of work developers would have to employ to implement a recommendation functionality, which would in turn allow developers to concentrate their efforts on their core application.

With the expansion of full-stack application development, a clear separation in terms of single responsibility principles has emerged to enable DevOps practices and agile methodologies in one project. Modern software architectures are departing from monolithic and tightly coupled approaches as they have shown to display limitations for maintenance, scalability, security, responsibility principles for DevOps and dependency issues (GOS; ZABIEROWSKI, 2020; FRANCESCO; LAGO; MALAVOLTA, 2018).

Based on the premises previously stated, this project aims to create a solution that is both widely available and simple to deploy, as well by using standard and state-of-the-art patterns that allow users to implement recommender systems services with much less effort than from scratch.

1.2 PROBLEM

New recommendation algorithms are frequently released, and they are frequently updated (ZANARDI; CAPRA, 2011). This is usually unimportant to users who want to use the recommendation functionality without interruptions in the development flow of an application or simply use the recommendation (in case the user is the end-user for a service and not a developer) and wants a readily available service. Because software development teams may not have personnel available to maintain the functionality, they may choose to use a third-party service to have the functionality implemented and maintained (RUDMARK, 2013). There are no known cases in the literature that completely address the issue, where viable, functional and free alternative services could provide additional functionality as a recommender systems within the loosely coupled service approach for major web applications, additionally, most recommender services available do not make use of algorithms and interfaces for both content based filtering and collaborative filtering, thus there is evidence that there is potential for pioneering proposals in the field of Recommender Systems as stand-alone Cloud Services.

During software development, usually there are concerns of feature separation and coupling. Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are (ISO/IEC/IEEE. . . , 2010). Tightly coupled systems tend to force a ripple effect of changes in other modules in case a change in made to one module, assembly of modules might require more effort and/or time due to the increased inter-module dependency, and a particular module might be harder to reuse and/or test because dependent modules must be included (PHAN, 2019).

If a company fails to provide a recommender system engine for its services (particularly those related to listing), it may be unable to provide a compelling user experience, which may generate a negative impact on their product's reception. As such, the company's overall potential market presence might be hindered by the lack of this feature. Additionally, a badly implemented or badly maintained recommender system may also result in bad user experience, a broken software environment, and a shift of development duties. In critical scenarios, the absence or bad implementation of recommendation services can lead to loss of revenue for companies. As such, a reliable and easy to maintain solution is necessary.

1.3 OBJECTIVE

This project tries to address the scarcity of non-commercial solutions for cloud-ready services that provide a Recommendation module as an additional feature to a major software or product that need a recommender system as a core feature. A containerized API solution will be developed. Overall, this work covers the following specific objetives:

- **SO1**: Conduct a comprehensive literature review on Recommender Systems, focusing on their concepts, modeling, techniques, evaluation forms, and practical applications, while also exploring optimal architectural styles for handling multiple calls with satisfactory response times;
- **SO2**: Investigate and implement web architectures that synergize with the principles of recommender systems for seamless integration into larger applications;
- **SO3**: Develop a recommendation service API that simplifies interactions for its consumers, sidestepping inherent complexities while offering both collaborative filtering and content-based filtering;
- **SO4**: Design and ensure the deployability of the API as a service, emphasizing its scalability and comprehensive documentation to aid developers.

1.4 RESEARCH QUESTIONS

In light of the problem and the objectives delineated, the research questions will guide the trajectory of the work. The answers to these questions will address facets of the problem presented. These questions are pivotal in validating the proposed solution. The following questions are intended to be addressed during the evaluation phase:

- **RQ1**: How do Web Architectures and Recommender Systems principles synergize to enhance software development, and which architectural style is optimal for a Recommendation System service?
- **RQ2**: What are the key features and functionalities that developers expect from a recommendation service API, and how can these be seamlessly integrated into larger applications?
- **RQ3**: What are the inherent complexities of recommendation systems, and how can an API be designed to simplify interactions for its consumers?
- **RQ4**: How does the developed API perform in terms of scalability, deployability, and utility, and what are the areas of improvement as identified by developers?

1.5 METHODOLOGY

This work investigates, proposes and evaluates a new software to provide an accessible Recommendation System core feature using state of the art methods. The methodology used to develop this project is as follows:

- **Literature Review**: Initially a literature review was conducted to learn about the state of the art of Recommendation, system architecture patterns, software APIs for web services and cloud services;
- **Research Opportunity Detection**: Through the research, it was identified the need of a dedicated and loosely coupled recommendation service core for larger web services;
- **Proposal Implementation**: Through the identified opportunity, a complete recommendation API was developed with a Microservice architectural approach always in mind, targeting to receive different types of data in content, goals and sizes;
- **Experiment Execution and evaluation**: Several test coverage principles were applied to the produced software API to fulfill availability and accuracy expectations, as well to meet with the proposed experiment objectives. Through the guidance of literature established metrics, this work attempted to solve most concerns regarding web APIs, particularly about usability, documentation, codebase maintainance, reliability and software availability.

1.6 OUT OF SCOPE

This project does not aim to provide, describe or suggest new Recommender System algorithms and paradigms. It is essentially a cloud-first API and architectural solution designed to replace a lengthy and often unreliable process of implementation of Recommender Services within other software projects as a third party and self contained service. This project also is not intended to be used as a commercial solution or a major maintained Software as a Service (SaaS) solution by itself, but rather be a reference for other SaaS solutions that can use the results of this work as a preliminary-step for other system designs.

1.7 DISSERTATION STRUCTURE

In this chapter we introduce the theme of the dissertation, we present the motivation and the problem that underlie this work. In addition, we address the objective, research questions, methodology, and expected contributions. The next chapters are organized as follows: Chapter 2 presents a literature review on Recommender Systems, demonstrating the concepts, modeling, techniques and their algorithms, as well as evaluation forms and industrial examples; Chapter 3 discusses Web Architecture concepts and brings research results related to software, systems and services that attempt to provide decoupled recommendation modules and how they relate to our work; Chapter 4 presents the overall project software architecture, going in depth about the design choices, technologies, libraries, algorithms and requirements; Chapter 5 evaluates the built API by making use of known metrics and methodologies, and presents the results obtained; Chapter 6 analyzes survey results on a RecSys API, evaluating user experience, challenges, and potential solutions; Chapter 7 presents the conclusion of the work, presenting the contributions, limitations and future work.

Chapter 2

RECOMMENDATION SYSTEMS

As already introduced, Recommendation Systems are used as a part of a larger system. In this chapter we cover a literature review on Recommendation Systems. Initially an introduction is presented, as well as the tasks and concepts that are widely applied. We then address ways of modeling user data as *feedbacks* and the formal representation of datasets. The recommendation techniques of Content-Based Filtering, Collaborative Filtering and Hybrid Filtering are presented, described and exemplified, as well as their processes, advantages and disadvantages. Furthermore, some recommender algorithms used in this work are also presented and formally described. We also present the evaluation protocols and the prediction and ranking metrics, which are widely used to understand the results of the systems. Finally, we present some successful commercial applications.

2.1 INTRODUCTION

Ricci, Rokach e Shapira (2011b) state that, "in its simplest form, personalized recommendations are provided as an ordered list of candidate items. To find the items that will make up that list, Recommender Systems try to predict which product or service will appeal most to users, based on historical preferences.

Resnick e Varian (1997) in his article entitled "Recommendation Systems" show that these systems support and enhance the recommendation process. Basically, what we have here are techniques and program tools for providing suggestions of items that are useful to a user. In these systems, "item" is a generic term used to denote what the system offers to a user. (RICCI; ROKACH; SHAPIRA, 2011b).

Criteria such as "personalized", "interesting" and "useful" are essential in differentiating Recommendation Systems with Information Retrieval Systems, Information Filtering or Search Engines. In a search engine, the system should return everything that matches the terms used in the search. However, search engines have begun to use techniques for entering user data into the search process (BURKE, 2002).

2.2 TASKS AND CONCEPTS

Recommendation Systems have goals that applications implement in common. Ricci, Rokach e Shapira (2011b) summarize some of these goals, which are: increase the number of items sold, sell more different items, increase user satisfaction, increase member loyalty, and better understand our users' desires. These objectives are adapted or reformulated according to the application domain. For example, an electronic store wants to sell more items as well as sell different items, however a movie streaming service does not sell the item, but wants its users to have contact with as many of its available items as possible.

Herlocker et al. (2004) present eleven tasks that Recommender Systems should implement: find some good items, find all good items, understand context, sequence recommendation, group recommendation, navigation between items, reliable recommendation, improve user profile, allow user expressiveness, help users, and finally influence other users.

The goals cited by Ricci, Rokach e Shapira (2011b) and the tasks described by Herlocker et al. (2004) are not the only points that recommender systems should tackle. McNee, Riedl e Konstan (2006) warn that only achieving the goal of recommending is not enough and that recommender systems should aim at other aspects. Thus, further research brought aspects such as: diversity in recommendation, surprise to the user with an unexpected item, fairness to their preferences, items that bring novelty, usefulness of the item to the user, coverage, among others. Figure 2.1 demonstrates what is necessary and/or desired for user satisfaction according to Silveira et al. (2017) who perform an overview of how to measure how good the recommendation system is.

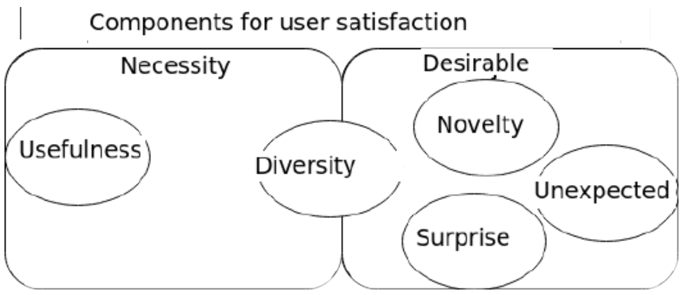


Figure 2.1 Components for user satisfaction (SILVEIRA et al., 2017).

According to Ricci, Rokach e Shapira (2011b) recommendation systems basically use three types of data to fulfill their purpose: items, users, and users' relationships with the items, called transactions.

- **Items**: These are the objects to be recommended. They can be characterized by their complexity, value and/or usefulness. The value of the item can be positive if it is useful to the user, or negative if it is inappropriate. Items can be represented using various representation and information approaches. In applications such as Netflixfootnotehttps://www.netflix.com, the item is the movie that will be recommended; in Spotifyfootnotehttps://www.spotify.com the item is a song; in

Courserafootnotehttps://www.coursera.org/ the item is a course; in Amazonfootnotehttps://www.amazon.com.br/ the item is a product (book, electronic, etc). In our work, formally the set of items in the system is represented by the letter I, where any one item is represented by the letter i or j;

- **Users**: They have various purposes and characteristics. To perform recommendation personalization, recommender systems use a set of data about the user. This data can be structured in various ways, and the selection of which data to use depends on the recommendation technique and the need of the application domain. In our work, formally the set of active users in the system is represented by the letter U, where any user is represented by the letter u or v;

- **Transactions**: These are records of the interactions between users and items in the system. They are structured data that store important information generated during the user's interaction with the system, which are used by the recommender algorithms to find useful recommendations. In our system, formally all transactions of users $u \in U$ with items $i \in I$ are represented by the letter T, where a transaction of a user u with an item i is represented as t_{ui}.

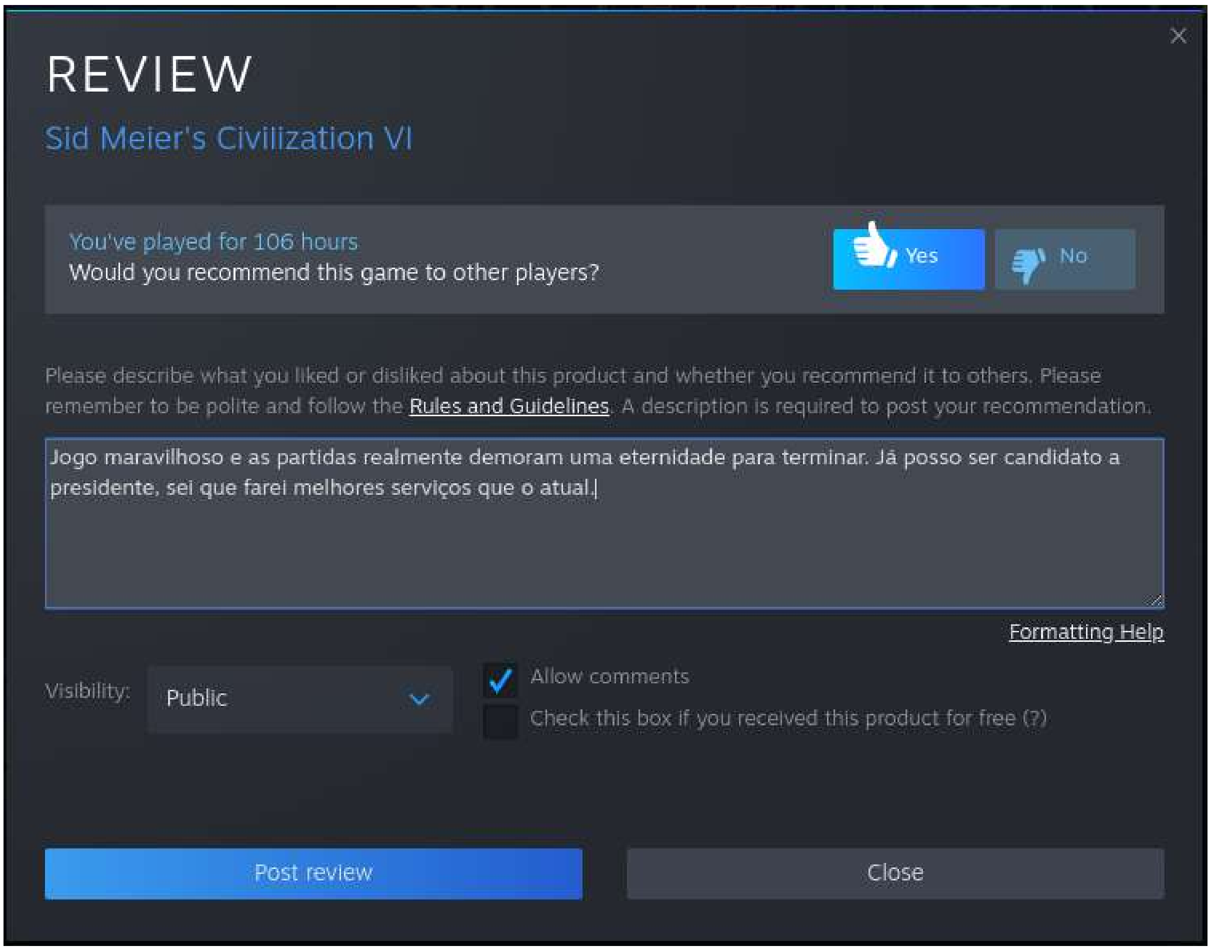
2.3 USER MODELING

The performance of the tasks of a recommender system is directly linked to the users and for this the user data must be properly modeled and structured. However, to model this data it is necessary that it is previously collected by the system. During the user's interaction with the system the transactions are recorded and serve as the database for the modeling. According to Ricci, Rokach e Shapira (2011b) it is possible to adopt two techniques to collect data from users about the items they interacted with, in most interactions users may interact positively with the item or interact negatively with the item. Isinkaye, Folajimi e Ojokoh (2015) present a third way to collect user feedback data.

2.3.1 Explicit Feedback

When the system requires direct feedback from the user in evaluating an item, asking the user about its relevance, this type of *feedback* is said to be explicit. This type of user feedback helps the system understand how interesting or relevant the item was to the user. Ricci, Rokach e Shapira (2011b), Celma (2008) present ways to get explicit *feedback* from the user. The use of binary *feedback* is one way to understand the relevance of the item to the user as the positive/negative (*like/dislike*). The use of discrete *feedback* with values starting from 0 opens up a greater possibility of evaluation about the interest of the user. One possibility for values is the Likert (LUCIAN, 2016) scale. Another way to get direct feedback from the user is to request to write comments or opinions about the item.

The application of the explicit *feedback* technique may require more than one type of simultaneous interaction from the user, for example, Figure 2.2 demonstrates that the Steam electronic game store requests both binary and comment feedback from the user.


REVIEW

Sid Meier's Civilization VI

You've played for 106 hours

Would you recommend this game to other players?

Please describe what you liked or disliked about this product and whether you recommend it to others. Please remember to be polite and follow the Rules and Guidelines. A description is required to post your recommendation.

Jogo maravilhoso e as partidas realmente demoram uma eternidade para terminar. Já posso ser candidato a presidente, sei que farei melhores serviços que o atual.

Visibility: Public

✓ Allow comments

Check this box if you received this product for free (?)

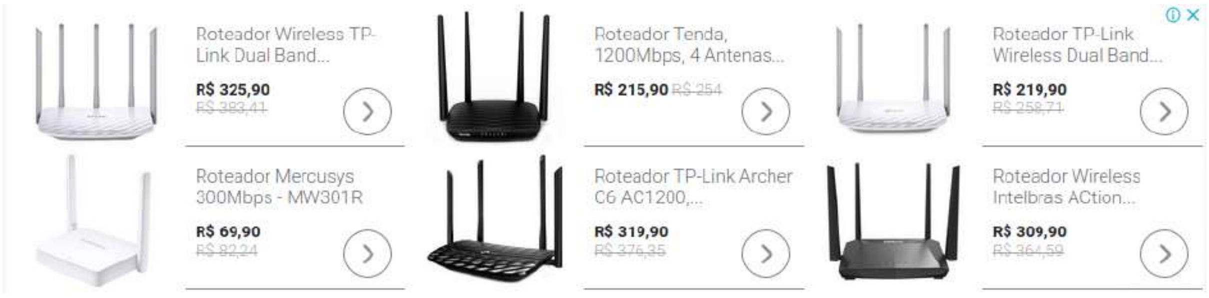
Figure 2.2 Feedback explicit binary and by comment.

2.3.2 Implicit Feedback

According to Celma (2008) “a recommender system can infer user preferences from passive monitoring of actions. Ricci, Rokach e Shapira (2011b) state that “implicit *feedback* does not require any action involving the user, taking into account that *feedback* is derived from monitoring and analyzing user activities”.

The implicit *feedback* method is based on assigning a relevance value to a user’s action on an item, such as: saving, discarding, printing, bookmarking, mouse movement, time on a page, among others. Thus, the amount of data that can be collected during the user’s interaction with the system is large. However, mistakes in the interpretation of data can happen, for example, when the user accidentally clicks on a Uniform Resource Locator (URL) or forgets that the music player is playing an album he doesn’t like.

Figure 2.3 demonstrates the recommendation of routers to the user, after the user performs a search for brands of routers. Thus, the recommendation system understood that during the next searches the user has the intention to search for more routers.



Roteador Wireless TP-Link Dual Band...
R$ 325,90 ~~R$ 383,41~~

Roteador Tenda, 1200Mbps, 4 Antenas...
R$ 215,90 ~~R$ 254~~

Roteador TP-Link Wireless Dual Band...
R$ 219,90 ~~R$ 258,71~~

Roteador Mercusys 300Mbps - MW301R
R$ 69,90 ~~R$ 82,24~~

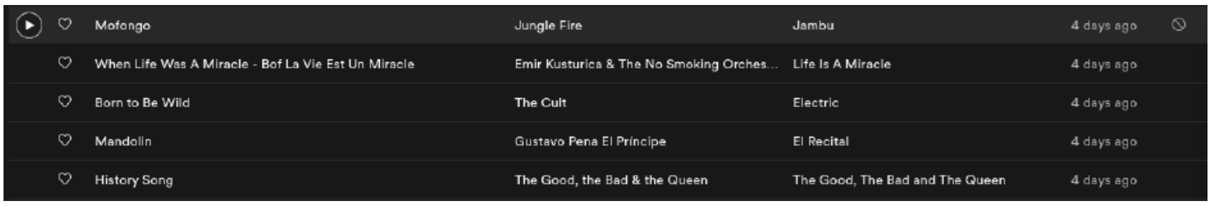
Roteador TP-Link Archer C6 AC1200,...
R$ 319,90 ~~R$ 376,35~~

Roteador Wireless Intelbras ACtion...
R$ 309,90 ~~R$ 364,59~~

Figure 2.3 *Feedback* implied from a textual search.

2.3.3 Hybrid Feedback

According to Isinkaye, Folajimi e Ojokoh (2015) the strength of both explicit and implicit *feedbacks* can be combined into a hybrid *feedback*, aiming to minimize the weakness and errors that each *feedback* may contain, thus enabling better system performance. Figure 2.4 demonstrates a hybrid *feedback*, where the user can give positive (heart button on the left) or negative (cancel button on the right) feedback, plus the system considers how many times the user has listened to a particular song.



Mofongo	Jungle Fire	Jambu	4 days ago
When Life Was A Miracle - Bof La Vie Est Un Miracle	Emir Kusturica & The No Smoking Orches...	Life Is A Miracle	4 days ago
Born to Be Wild	The Cult	Electric	4 days ago
Mandolin	Gustavo Pena El Principe	El Recital	4 days ago
History Song	The Good, the Bad & the Queen	The Good, The Bad and The Queen	4 days ago

Figure 2.4 Hybrid *Feedback*using the binary feedback and the feedback of how many times the song was heard.

2.3.4 Data representation

With the data collected and modeled, being *feedback* explicit or implicit, the system is able to start the phase of studying the data to find the recommendations. We present in Table 2.1 the formal representation of the structured data. The *feedbacks* are obtained from the system transactions, so they are modeled differently.

2.4 RECOMMENDATION TECHNIQUES

From the user preferences modeled by the application's *feedback* type, recommender systems apply techniques that aim to select the best items for their users. Each technique works in a specific way and has several recommender algorithms, which select different items to compose the lists. Thus, depending on the domain, besides studying the appropriate technique it is necessary to study which is the best algorithm.

The main recommendation techniques, which we describe during the next sections, are: Content-Based Filtering, Collaborative Filtering, and Hybrid Filtering. However,

Set	Set description	Relation	Relation description
U	Set of all users	u, v	Any user
I	Set of all items	i, j	Any item
T	Set of all transactions	t	Any transaction
t_u	All transactions from u	t_i	All transactions from i
t_{ui}	Transaction from u with i	$\|T\|$	Number referring to the size of the set T
$\|U\|$	Number referring to the size of the set U	$\|I\|$	Number referring to the size of the set I
R	Set of all valid *feedbacks*	$\|R\|$	Number referring to the size of the set R
$R(u)$	All user *feedbacks* u	$R(i)$	All item *feedbacks* i
$\hat{R}$	Set with all predicted *feedbacks*	$\|\hat{R}\|$	Number referring to the size of the set $\hat{R}$
r_{ui}	Value of user *feedback* u to the item i	$\hat{r}_{ui}$	Value predicted by the user's recommender u to item i
r_u	Value of all user *feedbacks* u	r_i	Value of all item *feedbacks* i

Table 2.1 Formal representation of the data sets used by the system.

these are not the only existing techniques. Burke (2002) does a survey of several existing recommendation techniques besides the three main ones, such as: demographic-based, utility-based and knowledge-based. Other studies also enhance the knowledge about the various recommendation techniques, such as Ricci, Rokach e Shapira (2011b) and Celma (2008). The Table 2.2 presented in the work of Burke (2002) is a summary on some recommendation techniques.

Techniques	Modeling	Input	Process
Collaborative	*Feedbacks* of all users	*Feedbacks* of the user being processed about the items of choice	Identifies similar users the user in processing and uses its *feedbacks* to recommend the items
Content	Item Data	User *Feedbacks* in processing about the items of choice	Shape a suitable classifier to its classification behavior and use it to select the items.
Demography	User demographic information and their *feedbacks*	User demographic information in processing	Identifies users who are demographically similar to the user being processed and uses its *feedbacks* to recommend the items
Utility	Item Data	A utility function over the items that describes user preferences being processed	Applies the function to the items and determines which items will be part of the ranking
Knowledge	Item Data. Knowledge of how these items meet the needs of a user	Description of needs or interests of the user being processed	Infers a correspondence between the item and the user in processing

Table 2.2 Recommendation techniques, their modeling, input and processing (BURKE, 2002).

2.5 CONTENT BASED FILTERING

One of the widely used techniques is Content-Based Filtering. This technique relies on the data that makes up the items to find similar items with user preferences. Adomavicius e Tuzhilin (2005) state that the content-based method has its origin in information retrieval research. According to Ricci, Rokach e Shapira (2011b) "recommendation systems that use content-based filtering attempt to recommend similar items with those that users have preferred in the past." Isinkaye, Folajimi e Ojokoh (2015) define that "the content-based technique is a domain-dependent algorithm and places more emphasis on analyzing the attributes of items to generate predictions."

The systems that implement the content-based filtering technique analyze the metadata that constitute the items, forming an information structure that is used during the recommendation process. This process is based on using the metadata of the items in a user's preferences to then find new items that have similar content and finally ranks these items in a recommendation list, taking into account how similar this new item is to the items in the preferences.

According to Lops, Gemmis e Semeraro (2011), "the recommendation process consists of comparing the attributes of the user's model with the attributes of an item. The result is a relevance judgment that represents the user's interest levels in that item."

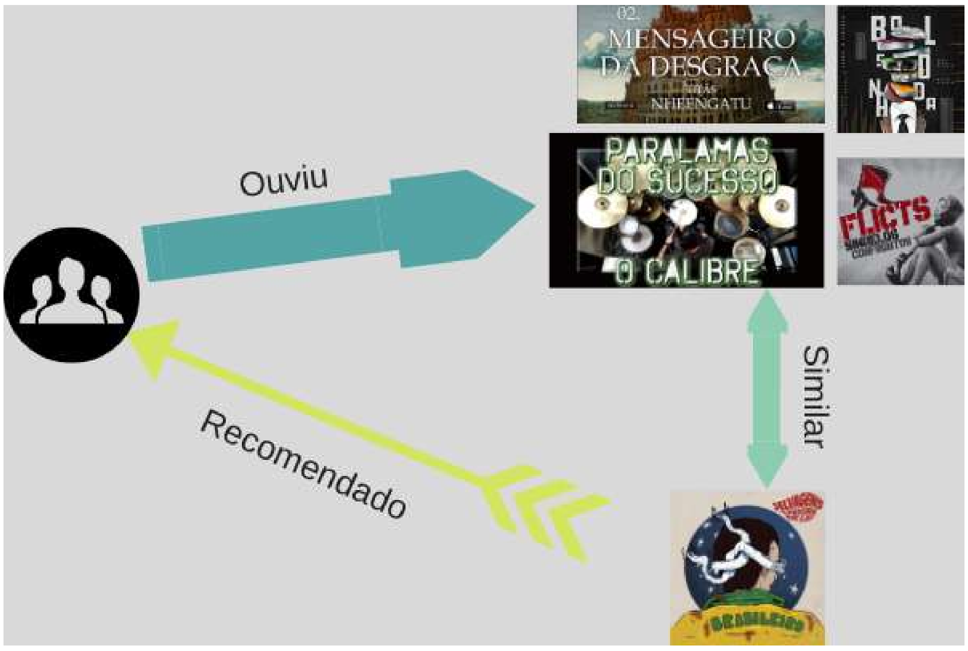


Figure 2.5 Example of a music recommendation based on the content describing the items.

As an example, in Figure 2.5 the user listened to four songs: "Bolso Nada" by Francisco el Hombre, "Mensageiro da Desgraça" by Titãs, "Desmascarando sua Bandeira" by Flicts and "O Calibre" by Paralamas do Sucesso; and was recommended with "Brasileiro" by Selvagens à Procura de Lei. This recommendation happens because the items have similarities in content, such as: musical genre, re-recordings in common, places and events of presentation, and participation in songs in albums in common.

2.5.1 Advantages and Problems

Recommendation systems that use content-based filtering have advantages and problems like all other techniques. The pros and cons of using the technique are presented in the studies of Isinkaye, Folajimi e Ojokoh (2015), Lops, Gemmis e Semeraro (2011), Adomavicius e Tuzhilin (2005).

The advantages of using this technique include:

- **Fast adaptation**: Adaptation of recommendations if user preferences undergo changes, whether those changes are large or small;
- **Small profile**: Recommending new items even if the user does not have many items in their profile or does not contribute a significant amount of *feedbacks*;
- **Privacy**: Because user profiles are analyzed separately without interference from other users' profiles. The technique keeps your users protected against attacks, for example the shill attack;
- **Security:** Depending on the system architecture it is possible to process each user separately, keeping the processing information only from a specific user, thus decreasing the chance of data leakage;
- **Explanability**: The steps performed by the algorithms to find the list of items are easy to audit and understand.

The technique also has disadvantages, which include:

- **Super specialization**: Recommended items have maximum similarity to the items in the user's profile, items outside of preferences typically do not enter the recommendation list, leading the user to live in a preference bubble;
- **Information dependency**: Items need metadata that describes them, the less information the more inaccurate the recommendation is, so item descriptions need to be well structured and data rich;
- **New user**: Even though the technique recommends items for profiles with few *feedbacks* when a user has no items in their profile, the system cannot find personalized recommendations. This problem is known as a cold start.

2.6 COLLABORATIVE FILTERING

One of the most widely used techniques in both research and industry is collaborative filtering. It takes the feedbacks of all users to recommend new items to another user, taking into account the user's preference history. The technique only takes into account the feedbacks and thus does not depend on the content of the items or extra information from the users. Ricci, Rokach e Shapira (2011b) state that "this approach recommends to an active user the items that other users with similar likes have liked in the past." Burke (2002) indicates that "collaborative recommender aggregates ratings or recommendation objects, recognizing commonalities among users by taking their ratings as a basis, and generating new recommendations based on inter-user comparison."

"It is likely that u's rating of a new item i will be similar to that of another user v, if u and v have rated other items similarly. Similarly, u is likely to rate two items i and j similarly, if other users have similar ratings for those two items." (DESROSIERS; KARYPIS, 2011).

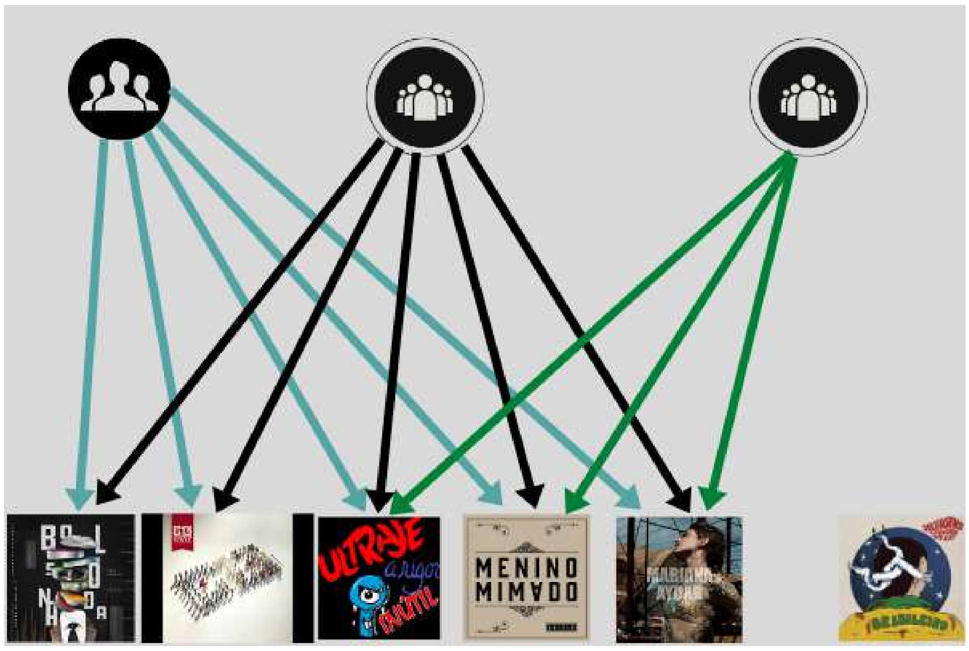


Figure 2.6 Example of music recommendation based on user preferences.

For example, Figure 2.6 demonstrates a possible recommendation using the collaborative filtering technique. The preferences shown are from three users about five songs, these being “Bolso Nada”, “Los Idiotas”, “Inútil”, “Menino Mimado” and “Tá?”. Two of the three users have positive ratings of all five songs and the third user does not know “Bolso Nada” and “Los Idiotas” but has similar ratings for the other three songs, so this third user will be recommended with these two songs. There is a sixth song in the figure called “Brasileiro”, however it does not enter into the recommendations, due to none of the users having positive ratings for it.

The algorithms of the technique are divided into two groups: memory-based method and the model-based method (RICCI; ROKACH; SHAPIRA, 2011b; ISINKAYE; FOLAJIMI; OJOKOH, 2015). A third method can be characterized, this one that sticks to basic ways of filtering and sorting the data. The algorithms are not model or memory based. In the following we describe the characteristics of each method:

- **Memory-based method**: In some studies it is called neighborhood-based method or heuristic-based method. It directly uses user ratings to find other neighbors that have similar ratings. This method can be implemented in two ways: i) user-based and ii) item-based. i) When implementing the user-based path the system calculates the similarity of the ratings between a user u and the other users, searching for users that have the highest degree of similarity. By finding the most similar neighbors (users), the items of their preferences are used as the basis of the recommendation to u. ii) The item-based algorithms search for similar ratings among the items, from a given item i in the preferences of u are found items that have similarity in the ratings. In Section 2.6.2 we present a user-based and an item-based variation of the K - Nearest Neighbors (KNN) algorithm;

- **Model-based method**: Uses machine learning algorithms to understand the user's profile. The profile learning is pre-computed and kept in the system which facilitates the recommendation process, however a learning period is required. The model-based method analyzes the user-item relationship to find new items. This method can work with sparsity better than the in-memory method. The learning methods used can be probabilistic, matrix factorizers, neural networks, among others;

- **Basic method**: Uses simple ways of filtering and sorting items. Some algorithms don't work with personalization, others just recommend items that the user hasn't had contact with yet, based on global values.

The Table 2.3 presents an overview of the methods, groups, and algorithms used in our work. In the next sections we present the algorithms in the Table, as well as the formal representations of each of them.

Methods	**Groups**	**Algorithms**
Basic	Basic	Popularity Best score
Based on memory	User	User-KNN
	Item	Item-KNN
Model-based	Grouping	Co Clustering
	Matrix Factoring	Singular Value Decomposition (SVD) SVD++ BMF
	Slope One	Slope One

Table 2.3 Methods, groups and algorithms of the collaborative filtering technique.

2.6.1 Basic

We will present two algorithms that are not memory or template based. These that are used in filtering and sorting items and are easy to implement and understand.

2.6.1.1 Popularity This is a widely used algorithm for sorting items and is commonly found in systems. This simple algorithm is based on counting how many profiles an item is present in, i.e. how many users preferred this item. The Equation 2.1 formally demonstrates the counting of items, for all items $i \in I$ it is counted how many times that item appears in the *feedbacks* ($|R(i)|$). Equation 2.2 formally demonstrates the recommendation, from the counted items the items that the user already knows are removed, thus keeping only the items unknown to the user.

$$popI = (\forall i \in I)|R(i)| \tag{2.1}$$

$$pop(u) = popI - R(u) \tag{2.2}$$

2.6.1.2 Best score Summarizes the *feedbacks* of users about a certain item, usually an average of the *feedbacks* values is taken. Thus, items with the best *feedbacks* that the user does not already know about are recommended. Equation 2.3 formally demonstrates that for every $i \in I$ an average of the *feedbacks* μ_i is performed. Equation 2.4 demonstrates that from the averaged items performed the items that the user already knows are removed, thus leaving only the unknown items.

$$countBS = (\forall i \in I)\mu_i \tag{2.3}$$

$$bs(u) = countBS - R(u) \tag{2.4}$$

2.6.2 KNN

Memory-based recommendation methods, also known as neighborhood-based, use the *feedbacks* of users, trying to find the degree of similarity between neighbors, the higher the similarity level the closer these neighbors are. This method is divided into two groups: the one that checks similarity between users and the group that checks similarity between items. One of the most widely used neighborhood-based recommenders is **KNN**. Desrosiers e Karypis (2011) and Koren e Bell (2015) present formulations of KNN for users and for items called User-KNN and Item-KNN.

2.6.2.1 User-KNN This implementation of KNN is based on finding users with the highest degrees of similarity. The relationship of all users $u \in U$ to all items $i \in I$ is modeled as a user-item matrix, where each row is a user and each column is an item. By selecting a user u to find recommendations it is checked which other users have similarities in feedbacks. User-KNN measures the degree of similarity between a user u and another user v by any distance measure. The following are some of the distance measures that are based on the calculation of distance between vectors.

$$sim(u,v) = PC(u,v) = \frac{\sum_{i \in I_{uv}}(r_{ui} - \mu_u)(r_{vi} - \mu_v)}{\sqrt{\sum_{i \in I_{uv}}(r_{ui} - \mu_u)^2 \sum_{i \in I_{uv}}(r_{vi} - \mu_v)^2}} \tag{2.5}$$

Equation 2.5 is the formal representation of Person correlation, two users u and v are given as input. In the equation I_{uv} represents the positions of the vectors which in turn are items. r_{ui} represents the *feedback* of user u on item i and r_{vi} of user v on item i. μ_u represents the average of the *feedback* values of user u and μ_v the average of user v.

$$sim(u,v) = cos(u,v) = \frac{\sum_{i \in I_{uv}} r_{ui} r_{vi}}{\sqrt{\sum_{i \in I_{uv}} r_{ui}^2 \sum_{i \in I_{uv}} r_{vi}^2}} \tag{2.6}$$

Equation 2.6 is the formal representation of cosine similarity, where two users u and v are given as input. In the equation I_{uv} represents the positions of the vectors of u and v which in turn are items i. r_{ui} represents the *feedback* value of user u on item i and r_{vi} of user v on item i.

$$\hat{r}_{ui} = \frac{\sum_{v \in N_i^k} sim(u,v) \cdot r_{vi}}{\sum_{v \in N_i^k} sim(u,v)} \tag{2.7}$$

Equation 2.7 formally represents the User-KNN prediction of user over an item $\hat{r}_{ui}$. Where N_i^k represents the k neighbors with the highest degree of similarity to user u. Thus, the recommender has two hyper parameters, the k and the distance measure.

2.6.2.2 Item-KNN This implementation of KNN is based on finding items with the highest degrees of similarity. The relationship of all items $i \in I$ by all items $j \in I$ is modeled as an item-item matrix, where each row is an item and each column is also an item. By selecting an item i to find the recommendations it is checked which other items have similarities in the *feedbacks*. Item-KNN measures the degree of similarity between an item i and another item j by any distance measure. The following are some of the distance measures that are based on the calculation of distance between vectors.

$$sim(i,j) = PC(i,j) = \frac{\sum_{u \in U_{ij}} (r_{ui} - \mu_i)(r_{uj} - \mu_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \mu_i)^2 \sum_{u \in U_{ij}} (r_{uj} - \mu_j)^2}} \tag{2.8}$$

Equation 2.8 is the formal representation of Person correlation, two items i and j are given as input. In the equation U_{ij} represents the positions of the vectors which in turn are users. r_{ui} represents the *feedback* of user u on item i and r_{uj} of user u on item j. μ_i represents the average of the *feedback* values of item i and μ_j the average of item j.

$$sim(i,j) = cos(i,j) = \frac{\sum_{u \in U_{ij}} r_{ui} r_{uj}}{\sqrt{\sum_{u \in U_{ij}} r_{ui}^2 \sum_{u \in U_{ij}} r_{uj}^2}} \tag{2.9}$$

Equation 2.9 is the formal representation of cosine similarity, where two items i and j are given as input. In the equation U_{ij} represents the users who have *feedbacks* of items i and j. r_{ui} represents the value of *feedback* of user u on item i and r_{uj} of user u on item j.

$$\hat{r}_{ui} = \frac{\sum_{j \in N_u^k} sim(i,j) \cdot r_{uj}}{\sum_{j \in N_u^k} sim(i,j)} \tag{2.10}$$

Equation 2.10 formally represents the prediction of user Item-KNN about an item $\hat{r}_{ui}$. Where N_u^k represents the k neighbors with the highest degree of similarity to item i. Thus, the recommender has two hyper parameters, the k and the distance measure.

2.6.3 Advantages and Problems

Recommendation systems that implement the collaborative filtering technique accept the advantages and disadvantages that exist in the approach. Some advantages are associated with neighborhood-based methods and some are associated with model-based ones, as well as for the disadvantages.

The advantages of using the techniques are:

1. **Simplicity**: Memory-based recommenders are easy to understand, simple to implement and adapt (ISINKAYE; FOLAJIMI; OJOKOH, 2015; DESROSIERS; KARYPIS, 2011; SU; KHOSHGOFTAAR, 2009);

2. **Justifiability**: The predictions of memory-based recommenders have easy and simple justifiability of why each item is being recommended (DESROSIERS; KARYPIS, 2011);

3. **Efficiency**: Memory-based recommenders do not require training to learn about user and peer profiles. Besides using only *feedbacks*, making it easier to process the data (DESROSIERS; KARYPIS, 2011);

4. **Performance**: Model-based recommenders have high recommendation accuracy (SU; KHOSHGOFTAAR, 2009);

5. **Stability**: The technique is not affected by instabilities in user profiles or momentary behavioral changes, thus adapting to high-impact changes (DESROSIERS; KARYPIS, 2011);

6. **Content-free**: Unlike the content-based filtering technique it does not become necessary data about the items to find recommendations, thus domains where information about the items is rare this technique gets better results (ISINKAYE; FOLAJIMI; OJOKOH, 2015; SU; KHOSHGOFTAAR, 2009);

7. **Case**: Unlike the content-based filtering technique approaches bring the possibility of items outside the user's bubble, avoiding overspecialization (ISINKAYE; FOLAJIMI; OJOKOH, 2015; DESROSIERS; KARYPIS, 2011).

The disadvantages of using the technique are:

1. **Enviesment**: The technique relies on *feedbacks* from others to find recommendations for similar new items. Using these *feedbabks* may influence the technique to find recommendations that do not match your preferences. If the user perceives that his preferences are not being respected and perceives the system as untrustworthy, he may abandon the system. One of the most common biases is the popularity bias, where the recommendations may be composed of the same group of items popular among older users of the system. Another bias is overlap, where a more preferred genre overlaps a less preferred genre;

2. **Cold start**: One of the main problems with the technique relates to when a user has just joined the application and has not contributed any *feedback*. In this case, the collaborative filtering technique cannot find recommendations for the user (ISINKAYE; FOLAJIMI; OJOKOH, 2015; SU; KHOSHGOFTAAR, 2009);

3. **New item**: A new item when registered in the application has *feedback* from the user, so the technique cannot find the item and add it to the recommendation list of some user who might come to prefer the item (ISINKAYE; FOLAJIMI; OJOKOH, 2015; SU; KHOSHGOFTAAR, 2009);

4. **Sparsity**: When only a small portion of users contribute *feedbacks*, causing a dearth of information. The effect of sparsity is a weak recommendation list, due to the difficulty of finding neighbors (ISINKAYE; FOLAJIMI; OJOKOH, 2015; SU; KHOSHGOFTAAR, 2009);

5. **Scalability**: Computation is usually done over a user-item matrix, where on small databases it gets good performance, but in the face of a large mass of data it can be inefficient (ISINKAYE; FOLAJIMI; OJOKOH, 2015; SU; KHOSHGOFTAAR, 2009);

6. **Synonym**: When similar items but with different names are registered in the system, they may not be related due to the difference in users' *feedbacks* (ISINKAYE; FOLAJIMI; OJOKOH, 2015; SU; KHOSHGOFTAAR, 2009);

7. **Gray Sheep**: Users who do not fit into any group or have preferences that do not allow finding neighbors do not receive recommendations, due to the technique not being able to find new items similar to preferences from *feedbacks* from others (QIN, 2013; SU; KHOSHGOFTAAR, 2009);

8. **Shill attack**: A competing item in recommendations may deliberately feed negative *feedbacks* into the system to punish an opponent, gaining advantages for themselves and not allowing users to receive the opponent's item as a recommendation (QIN, 2013; SU; KHOSHGOFTAAR, 2009);

9. **Privacy and Security**: Schafer et al. (2007) point out that "in centralized collaborative filtering architectures, a single repository stores all user ratings. If the central server becomes compromised, corrupted, or exposed, user anonymity can be destroyed." The same problem can happen in distributed architecture. Thus, the use of cryptography in the technique becomes necessary.

2.7 HYBRID FILTERING

Burke (2002) in their research elucidates that "a hybrid recommender system combines two or more recommender techniques to gain better performance with fewer disadvantages than any individual one." Ricci, Rokach e Shapira (2011b) explain that "a hybrid system combines techniques A and B by trying to use the advantages of A to fix the disadvantages of B."

Usually hybrid recommender systems combine the techniques of collaborative filtering and content-based filtering. However any technique can be combined as long as it meets the requirements of the application domain. By combining techniques the disadvantages are reduced from an advantage of another technique, so combining techniques that cover each other's disadvantages is best to achieve better accuracy.

There are several methods for combining recommendation techniques into a hybrid approach. Burke (2002) in his research maps out some possibilities for hybridization, which are described in Table 2.4.

Hybridization methods	**Description**
Weighted	The scores (or votes) of various recommendation techniques are combined to produce a single recommendation.
Alternated	The system switches between recommendation techniques depending on the situation.
Mixed	Recommendations from several different recommenders are presented at the same time.
Combination of characteristics	Resources from different sources of recommendation data are assembled into a single recommendation algorithm.
Cascading	One recommender refines the recommendations given by another.
Increased resources	The output of a technique is used as an input resource to another.
Meta-Level	The model learned by a recommender is used as input for another.

Table 2.4 Hybridization methods (BURKE, 2002).

2.8 SUMMARY

In this chapter we introduce the main concepts and tasks of a recommender system. We explain how to model user data. Next we address the recommendation techniques, discussing in detail each technique, addressing its concepts, algorithms, advantages and disadvantages. After the techniques we present ways to evaluate the recommendation systems, in their protocols and metrics. Finally, we present industrial applications of recommender systems. In the next chapter we will address the state of the art on the theme that this work aims to discuss.

Chapter 3

WEB ARCHITECTURE

This section intends to provide a comprehensive explanation on the proposed software's Web Architecture concepts based on the literature. Since the application is designed to be deployed on the cloud utilizing web protocols, it should follow a couple of standards and architectures, in order to enable communication between different peers (client and server) and allow interoperability between different systems; it should be able to use the default data transmission protocols such as Transmission Control Protocol (TCP)/Internet Protocol (IP), Hyper Text Transfer Protocol (HTTP) and Hyper Text Transfer Protocol Secure (HTTPS), and it can be displayed on representation formats (HyperText Markup Language (HTML), Cascading Style Sheets (CSS), JavaScript Object Notation (JSON)) as long as the API is in place, and everything should be accessible through the URL addressing standard.

3.1 SERVICE ORIENTED ARCHITECTURE (SOA)

Cordeiro (2012) defines Service-Oriented Architecture (SOA) as: “SOA is an IT philosophy that aims to facilitate integration between systems by guiding the creation and delivery of modular and loosely coupled solutions based on the concept of services”. Cordeiro (2012) describes SOA possessing the following characteristics:

- Services composition;
- Platforms abstraction and infrastructure technologies;
- Loose coupling;
- Use of patterns;
- Incentive to reuse components.

SOA enables modularity and splitting the complete environment into functional units, as there are complexities involved on the client-server model. The proposed software needs to be designed around the idea of distributed web services, particularly using SOA. Part of the role of SOA is developing a flexible and scalable application by utilizing loose coupling, heterogeneity and decentralized, where interoperability is a key feature. Modern SOA implementations currently heavily relies on Representational state transfer (REST) architecture, which is the basis of this work. SOA is a style of organizing (services), and Web services such as SaaS (Software as a Service) services, which this work is also characterized as, are its realization (BHOWMIK, 2020; KALE, 2018).

According to Cordeiro (2012), the use of SOA expects the following benefits:

- Ease of Maintenance: changes in business logic (implementation) do not affect existing applications;
- Reuse: new applications and processes (service consumers) can more easily reuse existing functionality;
- Flexibility: back-end systems and infrastructure can be replaced with less impact;
- Result: agility and cost reduction;
- Quality: guarantee of process homogeneity;
- Less time: agility in the impact analysis and evolutionary development of systems;
- Lower cost: reduced maintenance cost of the applications;
- Control: knowledge of existing assets.

This project will make use of the Microservices architecture, which is considered a variant of SOA. It will be better described on the Section 3.2.

3.1.1 Restful Architeture

REST is a web service architecture commonly used to create interactive applications. A Web service that follows REST architectural guidelines is called RESTful. A RESTful Web Service allows Web resources to be read and modified with a stateless protocol and a predefined set of operations. This approach enables interoperability between a client and a remote server on the Internet that provide these services. REST is an alternative to SOAP as way to access a Web service (FIELDING, 2000; BOOTH; HAAS; MCCABE, 2004).

REST uses key elements such as the resource URL, Request verbs, Request headers, Request body, Response body, and Response status codes (RUNGTA, 2020). The resources are the base URLs where the commands will be issued to the web server, the Request verbs are by standard POST, GET, PUT, PATCH, and DELETE, corresponding to create, read, update, and delete (CRUD) operations, the Request headers define the type of response required or the authorization details, the Request body is the data that is sent with the request, the Response body contains the data that was sent by the web server in return for the request, and the Response status codes are codes assigned to the response attributing information (100-199), success (200-299), redirection (300-399), client errors (400-499) or server errors (500-599) (RUNGTA, 2020; MOZILLA, 2021).

RESTful APIs are commonly described using languages such as the OpenAPI Specification (OAS) (MARTIN-LOPEZ; SEGURA; RUIZ-CORTéS, 2019), which provides a structured way to describe a RESTful API in a both human and machine-readable way, making it possible to automatically generate, for example, documentation, source code (clients and servers) and tests (MARTIN-LOPEZ; SEGURA; RUIZ-CORTéS, 2019).

Restful popularity is justified over its use of heterogeneous languages and environments, it enables web applications that are built on various programming languages to communicate with each other, it allows web applications to reside on different environments agnostic to operating systems, enables the programmer to code applications on different devices to interact with normal web applications, and web services programmed on the architecture based on REST services make the best use of Cloud-based services (RUNGTA, 2020).

In order to comply with characteristics required to be considered a RESTful web service, the server would provide the required functionality to the client, where the client sends a request to the web service on the server and the server would either reject the request or comply and provide an adequate response to the client. It also has a principle of statelessness, where it's up to the client to ensure that all the required information is provided to the server; the server should not maintain any sort of information between requests from the client, the client makes a request, the server outputs a response, the server will not store the previous question and new requests will get responses independently. In order to reduce server traffic, the client should store requests which have already been sent to the server in the format of cache and get the needed information instead of requesting it to the server again. REST also allow middleware layers between the client and the main server where extra services can be provided without disturbing the interaction between the client and the server. REST's interface / uniform contract exists on the HTTP layer by using the key verbs (typically POST, GET, PUT, PATCH, and DELETE) (RUNGTA, 2020). According to Mozilla (2020), the definition for each one of these key verbs are:

- **POST**: the POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server;
- **GET**: the GET method requests a representation of the specified resource. Requests using GET should only retrieve data;
- **PUT**: the PUT method replaces all current representations of the target resource with the request payload;
- **PATCH**: the PATCH method is used to apply partial modifications to a resource;
- **DELETE**: the DELETE method deletes the specified resource.

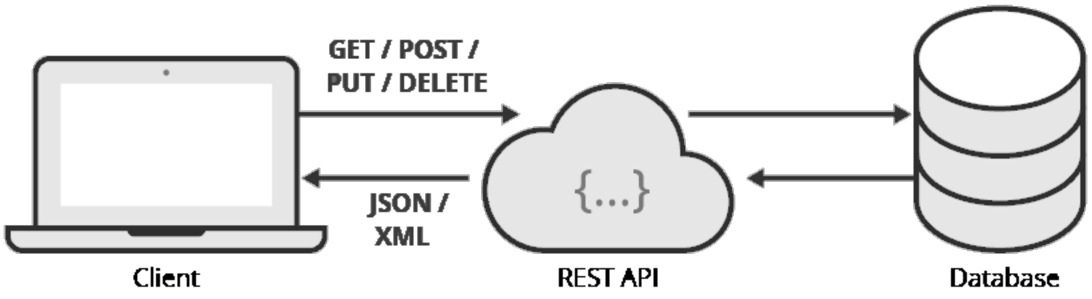


Figure 3.1 Example of the interoperability provided by the REST architecture.

This project will follow the API patterns as established by Zhang (2021), where it will have separate routes for users, items, feedbacks and all the REST actions (POST, GET and DELETE) to perform the backend operations. Further inspiration are projects that also utilized RESTful APIs for recommendation services, such as the work from García e Bellogín (2018) where it solely proposes an example implementation of a REST API focused on Recommender Systems, meeting the most typical requirements, while using a web client. For content-based filtering the API proposed by Teruya et al. (2020) should be used as a reference as the framework it used was capable of identifying the textual content of greatest interest to the user and recommending relevant related content. The work by Baldominos et al. (2015) designed a RESTful API for a recommendation system that implements both collaborative filtering and content-based filtering.

3.2 MICROSERVICE ARCHITECTURE

The Microservice Architecture is a variant of SOA which arranges an application into a collection of loosely coupled services, making use of fine-grained services and the protocols are lightweight, communicating over a network to fulfil a goal using technology-agnostic protocols such as HTTP (FOWLER, 2014; NEWMAN, 2015; WOLFF, 2017). Tooled around business capabilities (PAUTASSO et al., 2017), where services can be implemented using different programming languages, databases, hardware and software environment (CHEN, 2018), usually small in size and bounded by contexts, autonomously developed, independently deployable (NADAREISHVILI, 2016; CHEN, 2018) and characterized as being decentralized, built and released with automated processes (NADAREISHVILI, 2016).

According to Bettinger (2020), regarding the distinction between SOA and the Microservice architecture, “To put it simply, service-oriented architecture (SOA) has an enterprise scope, while the microservices architecture has an application scope.”, while adding that “In a microservices architecture, each service is developed independently, with its own communication protocol. With SOA, each service must share a common communication mechanism called an enterprise service bus (ESB). The ESB can become a single point of failure for the whole enterprise, and if a single service slows down, the entire system can be effected." and "Microservices architectures are made up of highly specialized services, each of which is designed to do one thing very well. The services that make up SOAs, on the other hand, can range from small, specialized services to enterprise-wide services".

The software will follow a Microservices architecture approach, it will not feature persistent database storage shared with the client and the software itself will be completely isolated from the client application, both sides won't interfere with its usage as long as the service contract isn't broken. It will operate by receiving JSON queries (where it could stream database information) and return a single JSON query to the client application containing the target list of recommended items(RIGHTBRAINNETWORKS, 2015; TECHCELLO, 2020; SONG et al., 2019).

3.3 CLOUD ARCHITECTURES

There are three cloud computing approaches: Software as a Service (SaaS), Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). This work is a Software as a Service framework. The approach used in this work follows the architecture of a SaaS system.

3.3.1 Software as a Service (SaaS)

SaaS is a service model utilized in Cloud Computing, where the users don't need physical hardware, nor buy extra software or install, maintain or update software. This service model improved businesses in terms of cost and time efficiency. The Google suite (Gmail, Google Docs, Google Sheets, Google Slides) is currently the prime example of SaaS, as it provides a cloud replacement to popular office software suites such as the Microsoft Office (KAVITHA; DAMODHARAN, 2020).

Another striking feature from SaaS is that it requires query processing, where it is performed query propagation and result propagation; the query propagation forwards the query through nodes, where it will then be processed by the main software, then once it finishes it performs the result propagation where the result is obtained by the client program. As many large applications are typically decomposed into functional primitives nowadays, SaaS proves to be useful, as the cloud services and its APIs are decoupled from major applications and its nature follows isolation and single-responsibility principles; it is especially useful for software utilizing SOA (KAVITHA; DAMODHARAN, 2020; SINGH et al., 2019, 2019; FRANKLIN; CHEE, 2019; BANERJEE, 2014).

Accordingly to INNVONIX (2019): "(...) SaaS is a means of delivering software as services over the internet to its subscribers, while SOA is an architectural model in which the smallest unit of logic is a service.", however, "(...) to get the maximum benefits of cost reduction and agility, it is highly recommended that enterprises integrate SOA and SaaS together".

3.3.1.1 Benefits

The SaaS approach also offers extra benefits, as when an interoperability with an application occurs, it lessens the use of the main application local server resource usage

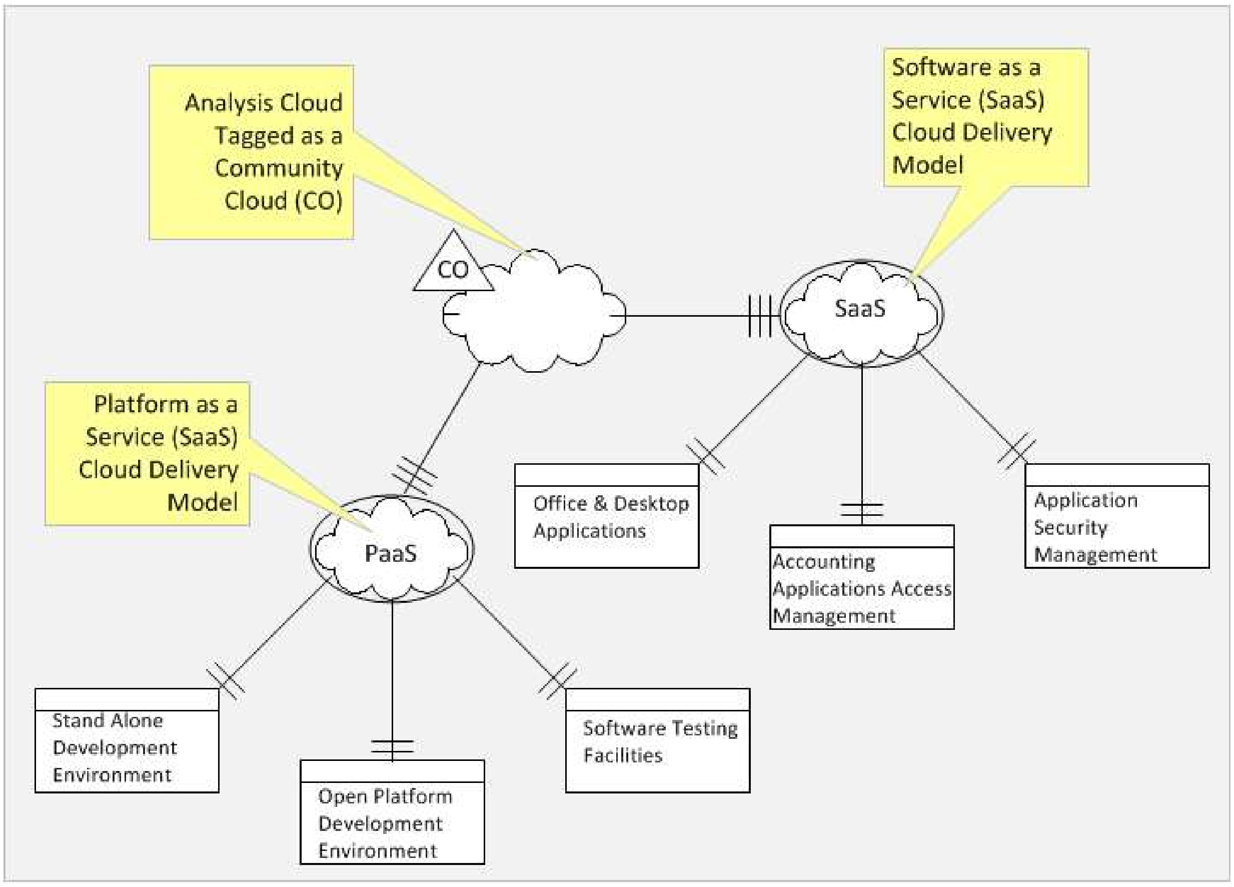

Analysis Cloud Tagged as a Community Cloud (CO)
Software as a Service (SaaS) Cloud Delivery Model
CO
SaaS
Platform as a Service (SaaS) Cloud Delivery Model
PaaS
Office & Desktop Applications
Accounting Applications Access Management
Application Security Management
Stand Alone Development Environment
Open Platform Development Environment
Software Testing Facilities

Figure 3.2 Cloud delivery model for SaaS and PaaS.

(instead using the remote server), and adds an extra layer of security as it doesn't provide direct access to databases, instead it just utilizes smaller samples of queries sent by the main application. Additionally, as it doesn't require installation, maintenance and updates, it doesn't require compatibility with software frameworks, operating systems, only requiring the end user to follow a set of required query standards when performing the interoperability. It might improve the QoS (quality of service) of the client application, as it offloads the raw hardware computability requirements, thus improving critical performance and response time, provided that the application is hosted on a server with superior specifications (GARBIS; CHAPMAN, 2021).

3.3.2 Platform as a Service (PaaS)

Platform as a Service, or PaaS, is a category of cloud computing services that allows users to provision, instantiate, run, and manage a modular bundle comprising a computing platform and one or more applications, exempting from the user the complexity of building and maintaining the infrastructure typically associated with developing and launching the application(s) (CHANG; ABU-AMARA; SANFORD, 2010; BUTLER, 2013). Thus, PaaS allows for higher-level programming with dramatically reduced complexity.

Accordingly to Watts e Haza (2019), “The delivery model of PaaS is similar to SaaS, except instead of delivering the software over the internet, PaaS provides a platform for software creation.”; examples of PaaS applications are AWS Elastic Beanstalk[1], Microsoft Azure[2], Heroku[3], salesforce.com[4], Google App Engine[5], Apache Stratos[6], and OpenShift[7].

This project is simply an API REST service. The user would be required to be responsible for handling the application in the could as well the data management, so it can’t be used as a complete PaaS solution.

3.3.3 Infrastructure as a Service (IaaS)

Infrastructure as a Service, or IaaS, is a category of cloud computing services where it’s provided high-level APIs while using low-level network infrastructure providing physical computing resources, usually providing pools of hypervisors (such as Oracle VM, KVM, Oracle VirtualBox, VMWare ESX or Hyper-V) within the cloud operational system that can support large numbers of virtual machines. IaaS can be used to provide all the infrastructure needed for PaaS and SaaS. The main objective of IaaS is to make it easy and affordable to provide resources such as servers, network, storage and other computing resources essential to build an on-demand environment, which may incorporate operating systems and applications (ANANICH, 2016; KNAPP, 2019; AMIES et al., 2012; MELL; GRANCE, 2011).

This project does not intent to include or cover any kind of implementation of virtual machines or any other kind of virtualization of physical hardware, thus it will not cover IaaS architectural goals.

3.4 RECOMMENDATION AS A SERVICE

Recommendation as a service is mostly an untapped potential for the industry. Many sites, services and platforms need to provide content personalization, at the same time, for both costs and time constraints, it is not feasible to implement a Recommender System implementation of their own. Therefore a system that performs Recommendation as a Service might be the solution. There are commercial and free options available for this end.

Among the commercial options, the most prominent are the recommender systems as services (RaaS). The basic idea of RaaS is to provide easy-to-use machine learning tools that allow even non-experts to train and deploy recommenders. The most well-known RaaS platforms are Amazon Personalize[8], Google Prediction API, and IBM Watson Machine Learning.

[1]https://aws.amazon.com/elasticbeanstalk/
[2]https://azure.microsoft.com/
[3]https://www.heroku.com/
[4]https://www.salesforce.com/
[5]https://cloud.google.com/appengine
[6]https://stratos.apache.org/
[7]https://www.openshift.com/
[8]https://aws.amazon.com/personalize/

	Amazon Personalize	Google Prediction API	IBM Watson ML
Price	$0.50/hour	$1.00/prediction	$0.10/prediction
User Friendliness	Easy	Some Coding Required	Requires Coding and Infrastructure Management

Table 3.1 Comparison of Amazon Personalize, Google Prediction API, and IBM Watson Machine Learning.

Amazon Personalize is a machine learning service that makes it easy to create personalized recommendations for customers using data from their applications. Google Prediction API is a cloud-based machine learning platform that makes it easy to build predictive models without having to code or manage infrastructure. IBM Watson Machine Learning[9] is a cloud-based platform that enables developers to easily build and deploy machine learning models.

Amazon Personalize is the most user-friendly platform, as it requires no coding or infrastructure management. Google Prediction API[10] is second, as it requires some coding but still provides a free tier for users to get started. IBM Watson Machine Learning is the least user-friendly platform, as it requires extensive coding and infrastructure management.

The Table 2.3 lists some of these services that attempt to fulfil the goal of providing recommendation services. The sources used for this Table were (RECOMMENDERSYSTEMS.COM, 2020) and (JENSON, 2020).

Not many free and open source SaaS could be found during the initial research; only two were found, and the only one being actively developed (Mr. DLib) has an emphasis on Digital Libraries, and does not use a generic approach this works intents to do. So there is no actively developed, mainstream RaaS platform being developed at the moment. However, there's a comprehensive list of free and open source Recommender System Frameworks available, as seen on Table 3.3 . The source used for this Table was (JENSON, 2020).

However, even though many of these frameworks would make a SaaS implementation possible, many of them have a Do It Yourself (DIY) approach, and are more applicable as PaaS since they don't have a cloud network infrastructure ready, and none of them have the following set of features in its integrity:

- High customization (algorithms, input settings, and data manipulation);
- Scalability;

[9]https://www.ibm.com/watson/developercloud/machine-learning.html
[10]https://cloud.google.com/prediction/docs/getting-started
[10]https://aws.amazon.com/personalize/pricing/
[10]https://cloud.google.com/prediction/pricing
[10]https://www.ibm.com/cloud-computing/bluemix/machine-learning

RaaS name	**Focus**	**Commercial**	**Actively developed**
Recombee	Generic	Yes	Yes
Darwin and Goliath	Generic	Yes	Yes
Mr. DLib	Digital Libraries	No	Yes
bX	Digital Libraries	Yes	Yes
BibTip	Digital Libraries	No	No
yuspify	e-Commerce	Yes	Yes
Kea Labs	e-Commerce	Yes	Yes
Sugestio	Generic	Yes	No
WebTunix	Generic	Yes	Yes
Peerius	e-Commerce	Yes	Yes
Strands	e-Commerce	Yes	Yes
SLI Systems	e-Commerce	Yes	Yes
ParallelDots	NLP	Yes	Yes
Amazon Personalize	Generic	Yes	Yes
Google Prediction API	Generic	Yes	Yes
Azure ML	Generic	Yes	Yes
Gravity R and D	Generic	Yes	Yes
Dressipi Style Adviser	Clothing	Yes	Yes
Sajari	Generic	Yes	Yes
IBM Watson	Generic	Yes	Yes
Segmentify	e-Commerce	Yes	Yes

Table 3.2 List of RaaS platforms/projects.

- Use of REST APIs;
- Support, in the same environment, both collaborative filtering and content-based filtering;
- Simple Deployment / Containerization (eg. Docker);
- Possibility for non-tech-savvy users to test recommendations.

3.4.1 Related work

Recommender Systems have been widely investigated in the literature, nevertheless deployment of Recommender Systems as Services are less frequent. In the following, we present some approaches in the literature.

A cloud based framework was proposed by Abbas et al. (2015) offering personalized recommendations about the health insurance plans. The plan information of each of the providers was designed to be retrieved using the Data as a Service (DaaS) architecture. The framework is implemented as Software as a Service (SaaS) to offer customized recommendations by applying a ranking technique for the identified plans according to the user specified criteria. While this referenced work emphasized on a single goal (a

Framework name	**Actively developed**
The Universal Recommender	Yes
PredictionIO	Yes
Raccoon Recommendation Engine	No
HapiGER	Yes
EasyRec	No
Mahout	Yes
Seldon	Yes
LensKit	Yes
Oryx v2	Yes
RecDB	No
Crab	Yes
predictor	Yes
surprise	Yes
lightfm	Yes
Rexy	No
QMF	No
tensorrec	Yes
hermes	No
spotlight	Yes
recommenderlab	Yes
CaseRecommender	Yes
ProbQA	Yes
Microsoft Recommenders	Yes
Gorse	Yes
Nvidia Merlin	Yes
Rcmmndr	No

Table 3.3 List of Open Source Recommender System Frameworks.

recommendation service for health insurance plans), this work intents to allow broader uses.

The work from García e Bellogín (2018) propose and show an example implementation for a common REST API focused on Recommender Systems, meeting the most typical requirements faced by Recommender Systems practitioners while being open and flexible to be extended. The work also presented a Web client that demonstrates the functionalities of the proposed API. This referenced work shares the intended focus on RESTful API design, albeit the current work will provide a full deployment environment as well full data manipulation to be standardized.

The work from Okon E Uko, B O Eke e Asagba (2018) designed and developed a recommendation model that uses object-oriented analysis and design methodology (OOADM), and improved collaborative filtering algorithm and developed an efficient quick sort algorithm to solve these problems. The similarity to this work is that the recommendation system was coded in Python, specifically the Django framework.

A proposal was done by Jain e Kakkar (2019) where a Job Recommendation System based on Machine Learning and Data Mining Techniques, while making use of a RESTful API and Android IDE. The referenced work work shall also be used as a reference to design a RESTful API and client-side features using the Android IDE as a template.

An API for web recommendation systems was developed by Teruya et al. (2020), composed of a Middleware and a Framework capable of identifying the textual content of greatest interest to the user and recommending relevant related content. The core recommendation system as well as the API from this referenced work will be used as a base particularly to content-based filtering. The current work differs in desired architecture as it is being designed to be a software beyond the API.

The work from Baldominos et al. (2015) describes the development of a web recommendation system implementing both collaborative filtering and content-based filtering, additionally the recommender system is deployed on top of a real-time big data architecture designed to work with the Apache Hadoop ecosystem, thus supporting horizontal scalability, and is able to provide recommendations as a service via a RESTful API. The performance of the recommender is measured, resulting in the system being able to provide dozens of recommendations in a few milliseconds in a single-node cluster configuration. The referenced work's metrics and benchmarking will be used as a template for the current project. It is similar in the regard that it uses both collaborative filtering and content-based filtering, but it is designed around a different ecosystem.

The author Ben-Shimon et al. (2014) provides as a demonstration an overview of a true backend system that allows a typical website owner to exactly perform the integration of a recommendation service, where the e-commerce retailer wants to have some control over the service, being able to configure the recommendation templates, turn off/on the service, apply filters on the recommendations, or define the feature templates; control the recommendation service in terms of configuration, management and monitoring. The referenced work was developed under a single scope (e-commerce) and that differs from this project, but it can be used as a reference particularly in regards to the customization of features.

The paper from Çano (2017) presents some of the most common recommendation applications and solutions that follow SaaS, PaaS or other cloud computing service models. They are provided from both academic and business domains and use recent data mining, machine learning and artificial intelligence techniques. According to the author, the trend of these types of applications is towards the SaaS service model which seems to be the most appropriate especially for enterprises.

Gorse, a project from Zhang (2021), is not available as a paper, but is an open source recommendation system written in Go, aiming to be a universal open source recommendation system that can be easily introduced into a wide variety of online services. By importing items, users and interaction data into Gorse, the system will automatically train models to generate recommendations for each user. As features, this project supports multi-way matching (latest, popular, collaborative filtering based on matrix factorization) and FM-based custom classification, performs single node training, distributed prediction, and ability to achieve horizontal scaling in the recommendation phase, also provides RESTful APIs for data and recommendation CRUD requests, and provides CLI tools for data import and export, model adjustment, and cluster status checking. The referenced work does not feature content-based filtering, something that is within the scope of the current work.

During research, we did not identify popularity on the RaaS field. This work intents to use a generalist approach within its Recommender System algorithm, in order to be usable by as many different use cases as possible, be it to provide functionality for eCommerce systems (where it could provide recommendations for products), to provide functionality for media indexation platforms (movies, music), or even specific commercial uses, such as tourism (travel/flight recommendations). The intent of this work is to require the least amount of effort as possible from the end user's system perspective, as it won't require installation, deployment, updates or maintain once the environment is already set up on a remote server; it will still require the user to follow guidelines, however, on how the queries should be properly standardized in order to make the interoperability work (well-defined service contracts). This recommender service will support both collaborative filtering and content-based filtering. Where the content filtering will be typically based on the user's preferences and interests, and the collaborative filtering will compare the user's preferences with the ones from other users. Additionally, the framework should be optimized enough to support large data sets, and be user-friendly for both developers and end users.

3.5 SUMMARY

This chapter addressed the Web Architecture principles that will be used as design references for this work, such as SOA, Microservices, REST architecture, and cloud architectures (SaaS, PaaS and IaaS) where SaaS in particular will be the targeted one, as the seen by the benefits this cloud computing approach can provide to the industry and its practical implementation. During the next chapters, we will discuss in-depth the strategies devised to fit this project in the proposed architectural concepts as well the designed workflow for the core Recommendation System.

Chapter

4

AN INTELLIGENT SELF-CONFIGURING RECOMMENDER SYSTEM AS A SERVICE

This section provides a comprehensive explanation on the proposed software solution's Web Architecture. The solution's API architecture will be three-tiered, with a client-server interoperability, and an extra layer of logic between them.

4.1 WEB SERVICE ARCHITECTURE

The core architecture will rely on the API that will be provided as a Microservice. In this work, it will be optitionally provided a Web Client where the end user will perform interactions that will then be interfaced on the backend with the core API. A Relational Database Management System (RDMS) database will also be needed to store the datasets. When the user interacts with the client, RESTful commands will then be sent to a layer of API Gateway, which would then be forwarded to the responsible Microservice, which in this case, would be the Recommendation System. When the service is executed, the first step will be validating the inputs, analysing the query, accepting if valid and rejecting if invalid, and if validated it will then proceed preparing the training data, executing the designated training algorithm against the trained model, and then execute the main recommendation algorithm which will compile the data that will be sent back to the client as a JSON response object.

The overall system architecture will be structured as shown in Figure 4.1.

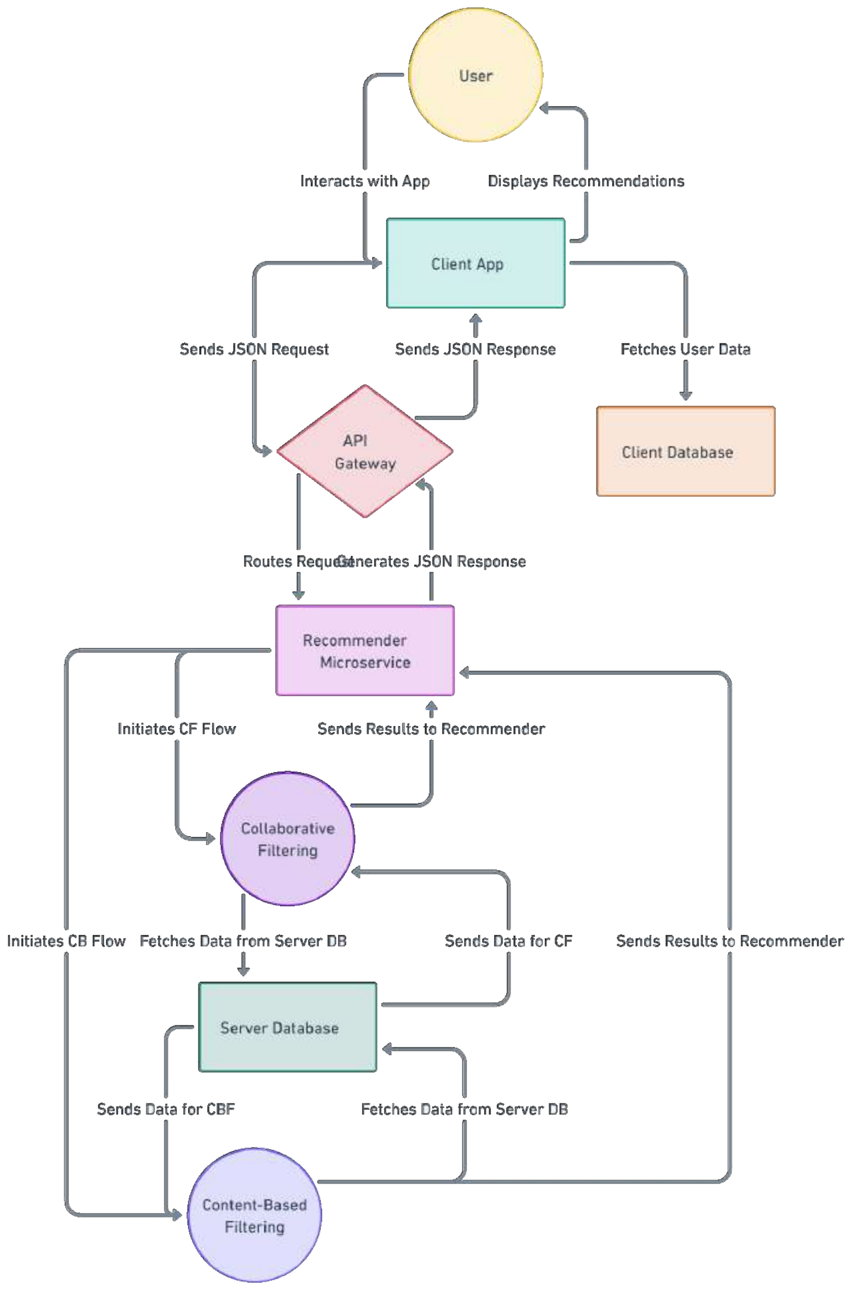


Figure 4.1 The complete Client to Service system architecture for this software.

4.1.1 The client side

First, we consider that the client that will eventually perform requests may have its own interface and database system, as well a selection of other services to query from. It is expected that this client shall retrieve data (either user provided or database provided) and then use this data as input to one of its services, namely a recommendation service.

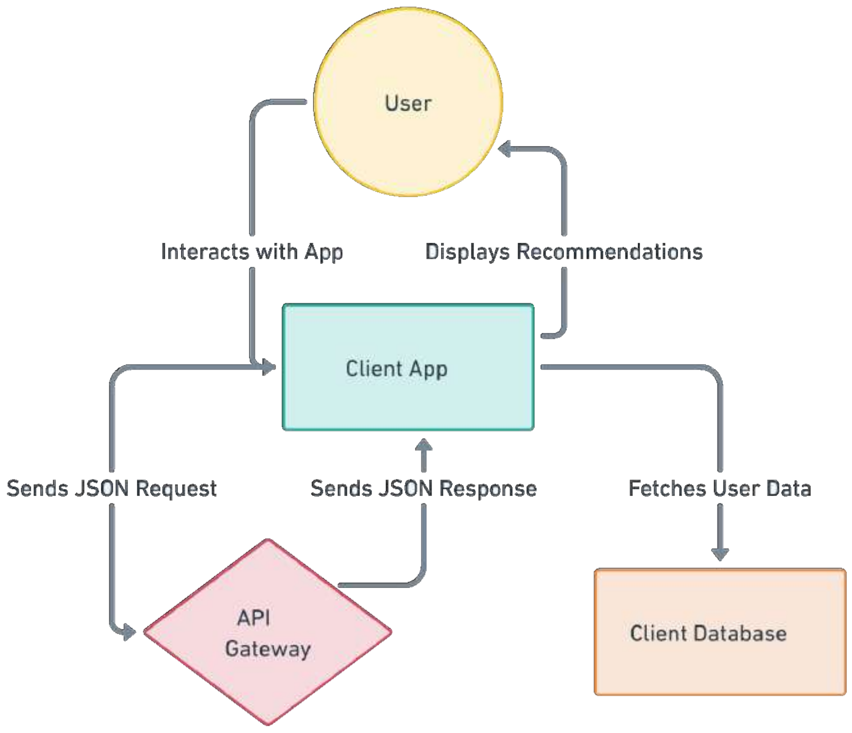


Figure 4.2 Client side architecture.

4.1.2 The Microservices side / API routing

Many applications make use of clustering and container orchestration (usually for Docker containers) such as Kubernetes, each container is designated for a separate service or feature. This is meant to avoid the monolithic software development approach, and thus break down the application into fine-grained services characterized by the Microservices architecture. On the case of this work, one of these Microservices has to be the complete environment for the Recommendation Service, which will receive JSONified data from the first step.

4.1.3 The Recommendation Service

As the last step, the Recommendation Service should receive the the data previously provided as input, designate it to the correct endpoint, perform an analysis on the input (data and format), and validate or invalidate it; if invalid, it should return a response message informing that the processing failed; if valid, it will continue the recommendation process. By continuing, it will then prepare the training data, start the training algorithm, generate a training model, execute the queried recommendation request on the recommendation algorithm, build a JSON object and then provide it as a response back to the client.

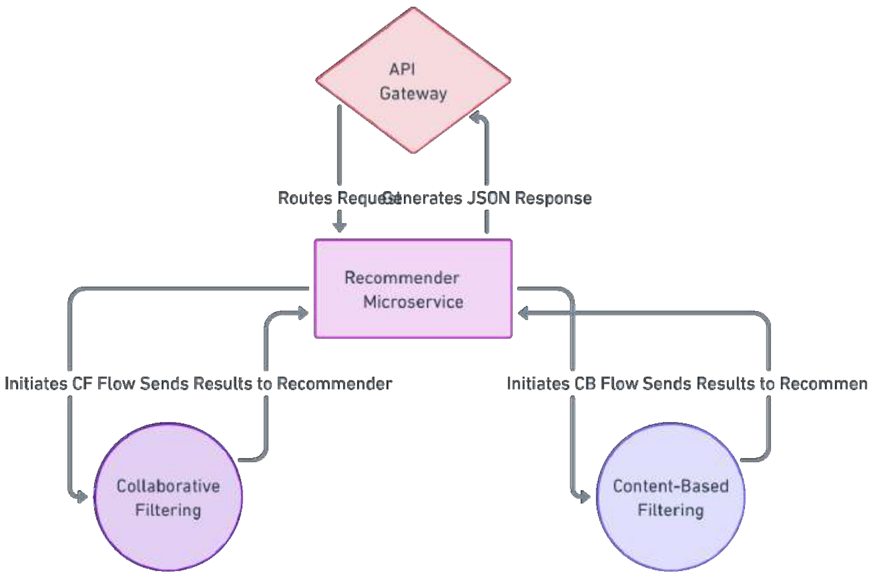


Figure 4.3 API routing layer bridging the Recommender System API.

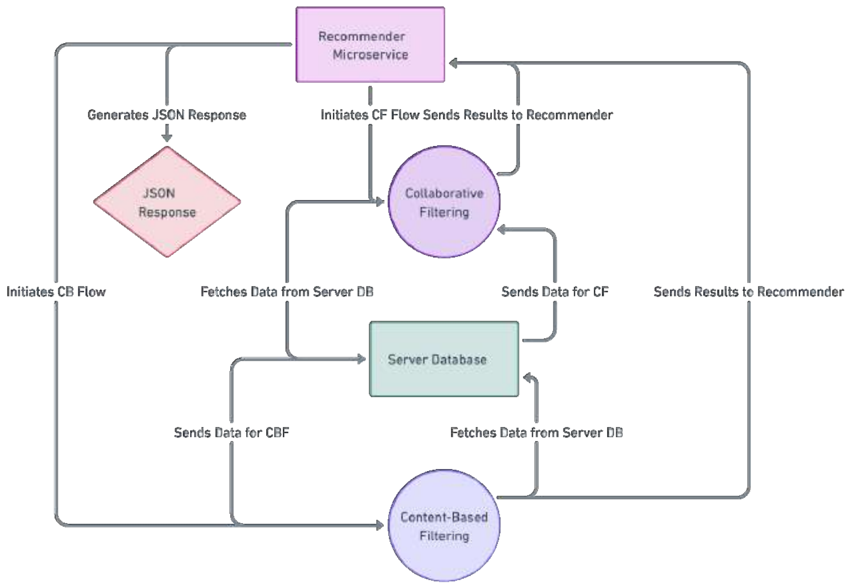


Figure 4.4 Microservice for the Recommendation Service.

4.2 REST API PATTERNS AND ROUTES

During the previous chapter, we saw references on how the industry software patterns can help shaping the proposed architecture, and how the REST API will be used to interface with the various components. The project will rely on a Microservice API, which will be interfaced with a web client. A RDMS database will be used to store the datasets. When the user interacts with the client, RESTful commands will be sent to a layer of API Gateway, which would then be forwarded to the responsible Microservice. This project will closely follow API patterns as established by previous relevant works, such as the work from (ZHANG, 2021) provides a pattern for REST API routes for the core Recommendation feature, for users, items and feedback (recommendation) data. These are particularly designed for collaborative filtering. It was identified that the work from (GARCíA; BELLOGíN, 2018) primarily focus on propose and show an example implementation for a common REST API focused on Recommender Systems, considering to meet the most typical requirements faced by Recommender Systems practitioners while being open and flexible to be extended.

Method	**URL**	**Description**
POST	**/item**	Add items to the database
DELETE	**/clear_db**	Clear the database
GET	**/docs**	Display the API documentation
POST	**/user**	Add user ratings to the database
GET	**/items**	Retrieve all items from the database
DELETE	**/user**	Delete a user from the database
GET	**/user**	Retrieve a user from the database
GET	**/users**	Retrieve all users from the database
GET	**/user/events**	Retrieve events related to a user
DELETE	**/item**	Delete an item from the database
PUT	**/item/{item_id}**	Update an item in the database
PUT	**/user/{user_id}**	Update a user's ratings in the database
GET	**/item/neighbors**	Retrieve similar items
GET	**/item/events**	Retrieve events related to an item
POST	**/event**	Add an event to the database
GET	**/events**	Retrieve all events from the database
GET	**/user/recommendations**	Retrieve user recommendations
GET	**/system**	Retrieve system information

Table 4.1 Revised table showing the updated API endpoints for the recommender system.

The resulting project API routes can be seen on the Table **??**.. The file called **___init___.py** creates a Flask application instance. The files **run.py** and **config.py** are respectively responsible for bringing up the application and the settings. In the folder **routes**, a file called **routes.py** is present. It contain all routes used as endpoints, without any API logic. The folder **models** contains files referencing data models by using Python classes representing a SQL table in the database, where attributes of a model translate to columns in a table. The **views** folder contains all of the application's main logic and database connection. The **apis** folder contains all of the auxiliary code related to table conversion and recommendation algorithms.

4.3 ALGORITHMS AND LIBRARIES

As seen in Chapter 2, according to Desrosiers et al. (2011) and Koren et al. (2015), there are two main types of memory-based recommendation methods: those that check similarity between users (user-KNN), and those that check similarity between items (item-KNN). Cosine similarity is one of the most widely used measures for determining similarity between vectors.The project makes use of approaches referenced by Sharma (2021) for both collaborative filtering and content-based recommendation. As Desrosiers et al. (2011) and Koren et al. (2015) explain, collaborative filtering is a method of making recommendations that is based on the similarity between users or items. Content-based recommendation, on the other hand, is a method that makes recommendations based on the similarity between the content of the items being recommended.

The following algorithms are used:

- Nearest Neighbors (collaborative filtering);
- Cosine similarity (content based recommendation).

This projects makes use of the following Python Libraries:

- Pandas (designed for data manipulation and analysis, used particularly to handle data for collaborative filtering);
- Scipy (used to perform algebraic manipulation of matrices, used particularly to handle item features pivoted from ratings during the collaborative filtering process);
- Scikit-Learn (used for its machine learning classification, regression and clustering algorithms, both during collaborative filtering and content based recommendation);
- SQLAlchemy (Object Relational Mapper that gives application that grants flexibility for SQL databases).

4.4 USED TECHNOLOGIES

This project makes use of the Flask Python Web Framework and its bundled Werkzeug toolkit for WSGI (Web Server Gateway Interface) applications where it's realized software objects for request, response, and utility functions. Flask natively supports RESTful request dispatching.

WSGI is used for the proposed software, which is a calling convention for web servers to forward requests to web applications or frameworks written in the Python programming language. WSGI has the server/gateway side, often running full web server software such as Apache or Nginx, or is a lightweight application server that can communicate with a webserver, and the application/framework side, where it's Python callable, supplied by the Python program or framework.

4.5 RECSYS AS A SERVICE

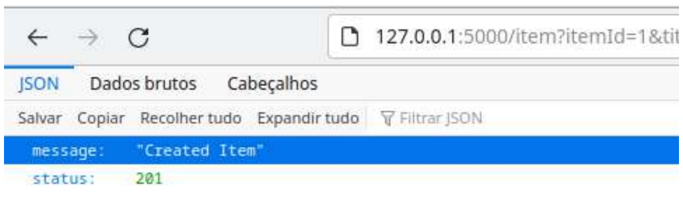
In this section, we will show examples of how the API Recommender System works. The first example is a call from a browser. The second example is a call from Postman (which would have the same effect as calling from a CLI client). The default browser REST verb to access pages is GET; while it is not ideal to use GET to register elements into the database, in the following example (that would be POST), we used to demonstrate the API's ability to receive data in three ways: by sending URL parameters, by sending JSON objects as POST to the designated endpoint, and by sending CSV files (also by POST) to the designated endpoint. In the following examples, we are going to add movie items to a relational database by querying the API in three different ways, using the same endpoint, which is the **base URL + /item**.

Considering http://127.0.0.1:5000/ as the localhost, a sample item (the movie Toy Story) was added to the database using the URL seen in the Figure 4.5. The the API returns a JSON object informing whether the new entry was successfully created or not, as seen in the Figure 4.6.

The other method takes as input a JSON object specifying the item, using POST, seen in the Figures 4.7 and 4.8. Lastly, the API also accepts to receive the items as a CSV spreadsheet, which will in turn be parsed for insertion into the database, as verified in Figure 4.9.

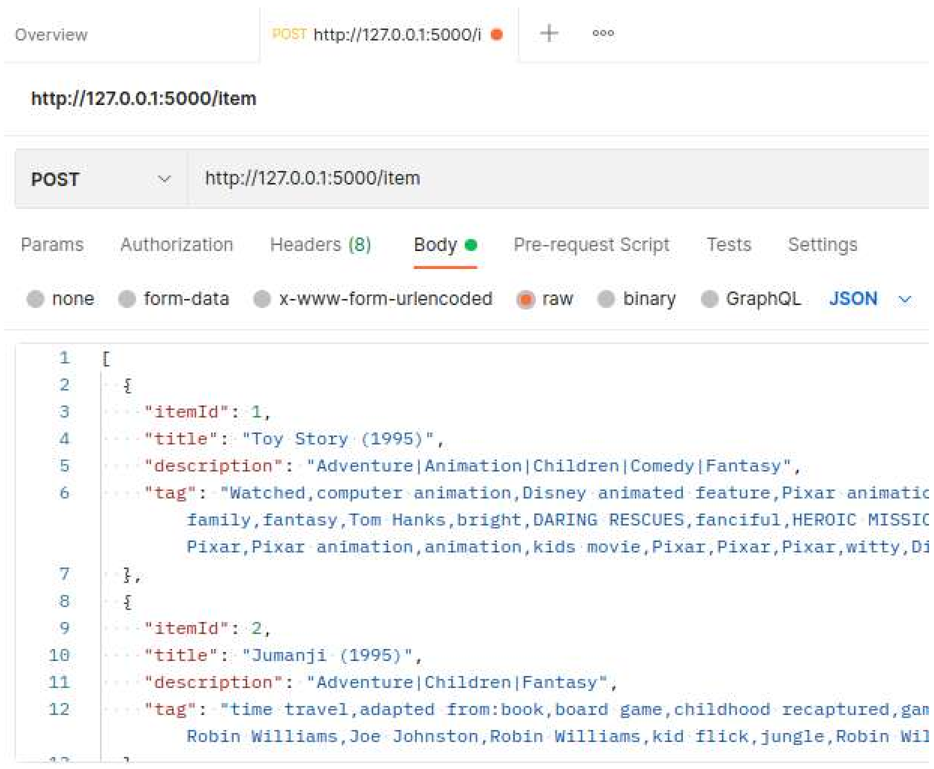
```
http://127.0.0.1:5000/item?itemId=1&title=Toy+Story &description=Adventure|Animation
|Children|Comedy|Fantasy ,&tag=Watched,computer+animation , Disney+animated+feature,Pixar+animation
,Pixar,animation,family,Tom+Hanks,Pixar,witty,Pixar,adventure,animated,animation,clever,comedy
,computer+animation,family,fantasy,Tom+Hanks,(...),Disney,Tim+Allen
```

Figure 4.5 Example of parameters being added to an URL endpoint, in this case, to add an item.



```
message:   "Created Item"
status:    201
```

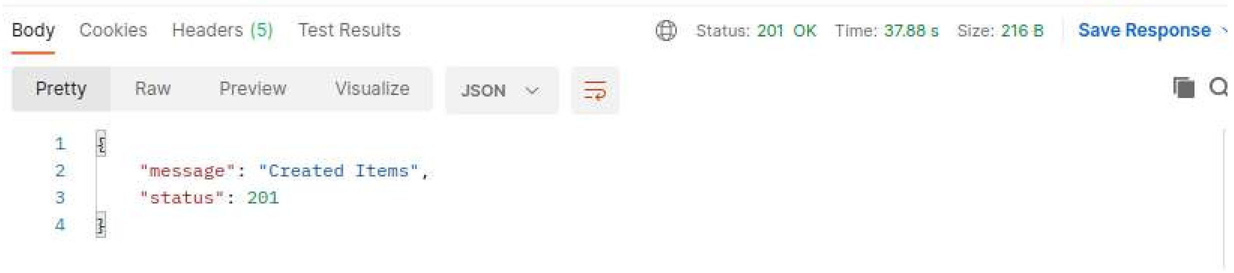
Figure 4.6 Sending items to be added to the main database as URL parameters.



POST http://127.0.0.1:5000/item

```
[
  {
    "itemId": 1,
    "title": "Toy Story (1995)",
    "description": "Adventure|Animation|Children|Comedy|Fantasy",
    "tag": "Watched,computer animation,Disney animated feature,Pixar animatic
        family,fantasy,Tom Hanks,bright,DARING RESCUES,fanciful,HEROIC MISSIC
        Pixar,Pixar animation,animation,kids movie,Pixar,Pixar,Pixar,witty,Di
  },
  {
    "itemId": 2,
    "title": "Jumanji (1995)",
    "description": "Adventure|Children|Fantasy",
    "tag": "time travel,adapted from:book,board game,childhood recaptured,gam
        Robin Williams,Joe Johnston,Robin Williams,kid flick,jungle,Robin Wil
```

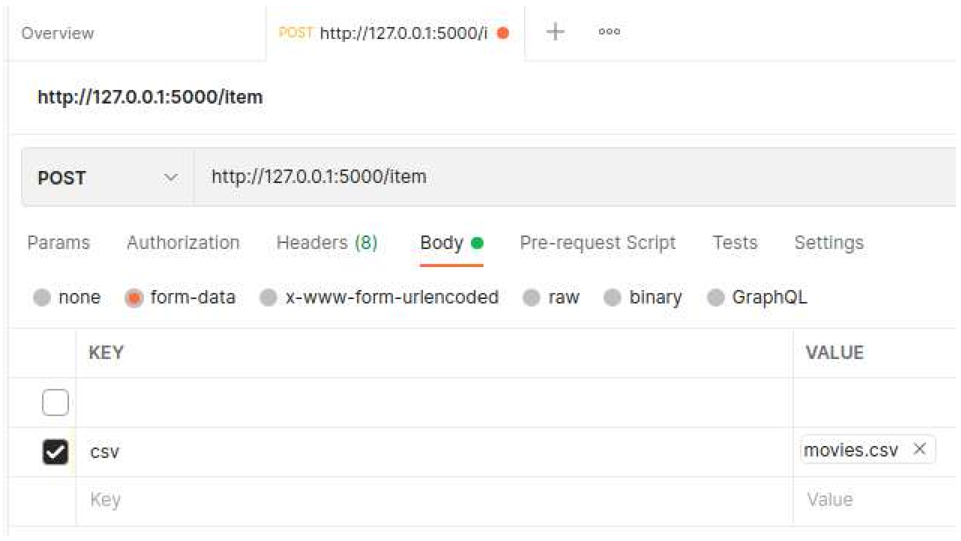
Figure 4.7 Sending items to be added to the main database as a JSON object.



Status: 201 OK Time: 37.88 s Size: 216 B

```
{
    "message": "Created Items",
    "status": 201
}
```

Figure 4.8 Response object.



POST http://127.0.0.1:5000/item

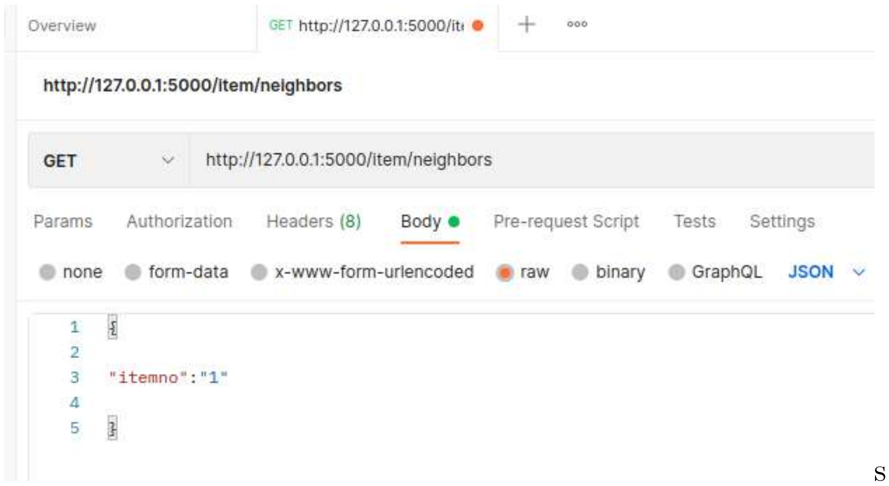
Body: form-data

KEY	VALUE
csv	movies.csv

Figure 4.9 Sending items to be added to the main database as a CSV file.

4.5.1 Default Recommendation routine

In order to perform the recommendations, a parameter should be passed identifying the desired item that needs to get similar items recommended. In the following scenario, we are retrieving similar items to the movie Toy Story, which we know it‘s the item number one in the database, using its ID number, in the Content-based filtering endpoint (ending in **/item/neighbors**) using the GET parameter. The API then returns a JSON containing information about the target element, and a list of recommended items with attributes of each one, which are position (by similarity), the item’s name, the similarity score, the item‘s description, and the tags (which are considered for recommendation in this particular case for content-based filtering).



http://127.0.0.1:5000/item/neighbors

GET http://127.0.0.1:5000/item/neighbors

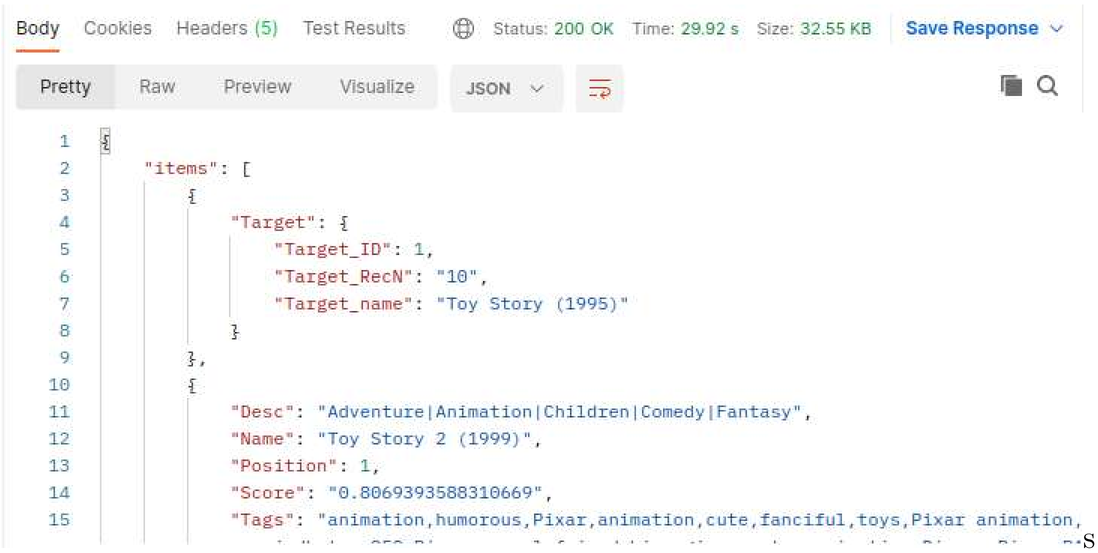
```
{

"itemno":"1"

}
```

S

Figure 4.10 The request JSON object.



Status: 200 OK Time: 29.92 s Size: 32.55 KB

```
{
    "items": [
        {
            "Target": {
                "Target_ID": 1,
                "Target_RecN": "10",
                "Target_name": "Toy Story (1995)"
            }
        },
        {
            "Desc": "Adventure|Animation|Children|Comedy|Fantasy",
            "Name": "Toy Story 2 (1999)",
            "Position": 1,
            "Score": "0.8069393588310669",
            "Tags": "animation,humorous,Pixar,animation,cute,fanciful,toys,Pixar animation,
```

S

Figure 4.11 The response JSON object.

4.6 SUMMARY

In this chapter, we discussed the proposed software's Web Service Architecture, describing its REST API patterns and routes, the algorithms and libraries employed, the project structure, and the expected system requirements. Furthermore, we discussed related works with similar goals, and we emphasized how this work deviates from them as an alternative and how it intents to provide new solutions under a different implementation method.

Chapter

5

EXPERIMENTAL EVALUATION

In this section, we are going to present the experimental evaluation of the proposed API architecture from the project based on guidelines established by the literature. Evaluation plays an instrumental role on each phase of software development, as it should be used to aid the system's design to be better prone to bugs, as well as improving user experience on later stages. However, the design phase is the most appropriate time to take into account the API usability issues (ZIBRAN; EISHITA; ROY, 2011). APIs that are developed with its users in mind and providing good usability encourage users to continue using them; users easily leave the APIs that they are not comfortable with (RAUF; TROUBITSYNA; PORRES, 2019). Well-thought-out APIs encourage programmers to use them productively and satisfactorily.

5.1 METHODOLOGY

We tested the API on different datasets in order to see how it would perform under different conditions. The chosen Dataset was the movie dataset from the project MovieLens, by GroupLens called ml-20m. This dataset contains information on 20 million ratings and 580,000 tag applications applied to 27,000 movies by 138,000 users. We used different hardware in our testing, including both high-end and low-end hardware. We also tested the API under different load conditions and under different recommendation conditions. The system was first tested on datasets with different amounts of items. The recommender system was asked to recommend 3, 5, and 10 items. The ml-20m dataset proved to be too large for the lower end hardware we tested on the cloud services, so different dataset sizes were evaluated, in order to get a better overview on the critical functionality of the API. The first environment condition that was varied was the type of hardware that the API was tested on. The API was tested on both high-end and low-end hardware. The second environment condition that was varied was the load condition. The load condition was varied by changing the complexity of the requests to the API. Another defined condition was the recommendation condition. The recommendation condition was varied by changing the number of items that the recommender system was asked to recommend.

5.2 TEST ENVIRONMENTS

In order to evaluate the performance of the recommender system API, we tested it on different datasets and under different conditions. We used the MovieLens dataset for our experiments, and varied the size of the dataset and the hardware used. We also tested the API with both collaborative filtering and content based filtering. We have deployed the API it in two ways: locally and server-side using different cloud services. The API was deployed and tested on popular Python cloud services such as **Heroku** and **PythonAnywhere** (under the Free plans). The tests were executed locally in two ways: by exclusively executing the two recommendation algorithms (content based and collaborative filtering) and also by running a RESTful API endpoint that performs the recommendation.

5.2.1 System specifications

The system specifications for each of the environments can be seen on the Table 5.1.

	Memory (RAM)	CPU Cache	CPU clock
PythonAnywhere	3 GB (limited from 16 GB)	35.75MB L3 cache	2.50 GHz
Local machine	16 GB	12 M	4.50 GHz
Heroku	512 MB	Unknown	Unknown*

Table 5.1 System specifications.

5.2.2 System requirements

The recommended hardware system requirements for this project can be seen on the Table 5.2.

System requirements			
CPU	RAM	Storage	Network
7th generation Intel Core i7 processor or better	16 GB	150 GB of free space	100 Mbps

Table 5.2 System requirements.

The software requirements for this project can be seen on the Table 5.3.

[0]* Said to be 1x-4x "compute".

Software Requirements			
OS	Database	Python version	Python libraries
At least Ubuntu 20.04 LTS	A RDBMS such as PostgreSQL	Python 3.6	All Python libraries from the requirements.txt file

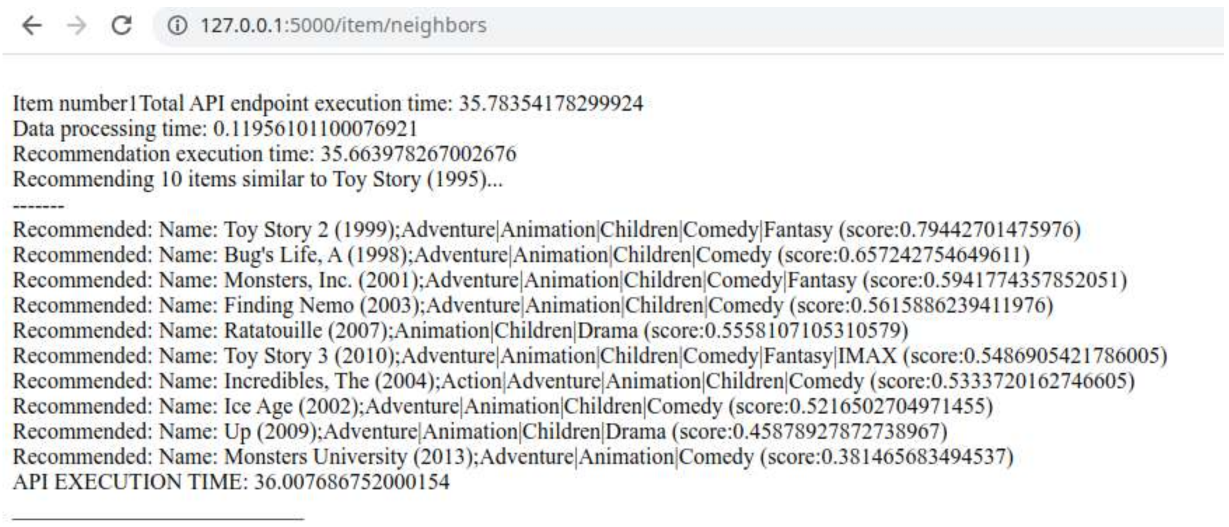
Table 5.3 Software requirements.

5.3 MEASUREMENT METRICS

An API can be considered useful when it correctly provides the desired functionality, and efficiency in terms of performance (regarding resource consumption, speedup, and so on) (ZIBRAN; EISHITA; ROY, 2011). For many competing goals (e.g., lower development cost, market demand), API designers take into account design criteria such as modularity, reusability, and evolvability, which mainly benefit those who are involved in development and maintenance of the APIs (ZIBRAN; EISHITA; ROY, 2011).

Our code extensively measured the I/O time for the API Endpoints by setting timers hardcoded through each task of the recommendation, in order to better measure the latency and response times. The performance of the recommender system is measured by the following metrics:

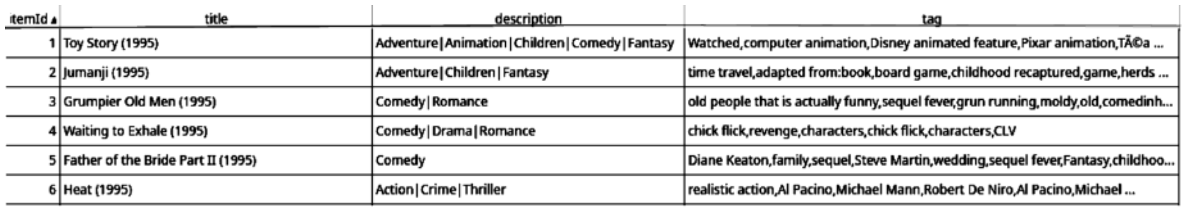
- **Filtering type:** Collaborative and Content-based;
- **Execution environment:** Locally (only the recommendation algorithm), Locally (API), PythonAnywhere (API) and Heroku (API);
- **Number of recommended items:** the number of items on the recommendation output;
- **Dataset size:** the amount of rows taken into account for the filtering type;
- **Total API endpoint execution time (seconds):** the total amount of time elapsed when the REST endpoint is executed from start to finish;
- **Endpoint execution latency:** the amount of time needed to execute only the REST endpoint (disregarding the recommendation);
- **Recommendation execution:** the amount of time that was needed to execute the core recommendation until the result is achieved;
- **Data processing time (seconds):** the amount of time that was required to process the data from the database and prepare it for the recommendation task;
- **Recommender algorithm execution time (seconds):** the elapsed time to find the recommended items using the recommender algorithm, with the data ready.



```
127.0.0.1:5000/item/neighbors

Item number1Total API endpoint execution time: 35.78354178299924
Data processing time: 0.1195610110007692l
Recommendation execution time: 35.663978267002676
Recommending 10 items similar to Toy Story (1995)...
-------
Recommended: Name: Toy Story 2 (1999);Adventure|Animation|Children|Comedy|Fantasy (score:0.79442701475976)
Recommended: Name: Bug's Life, A (1998);Adventure|Animation|Children|Comedy (score:0.65724275464961l)
Recommended: Name: Monsters, Inc. (2001);Adventure|Animation|Children|Comedy|Fantasy (score:0.5941774357852051)
Recommended: Name: Finding Nemo (2003);Adventure|Animation|Children|Comedy (score:0.5615886239411976)
Recommended: Name: Ratatouille (2007);Animation|Children|Drama (score:0.5558107105310579)
Recommended: Name: Toy Story 3 (2010);Adventure|Animation|Children|Comedy|Fantasy|IMAX (score:0.5486905421786005)
Recommended: Name: Incredibles, The (2004);Action|Adventure|Animation|Children|Comedy (score:0.5333720162746605)
Recommended: Name: Ice Age (2002);Adventure|Animation|Children|Comedy (score:0.5216502704971455)
Recommended: Name: Up (2009);Adventure|Animation|Children|Drama (score:0.45878927872738967)
Recommended: Name: Monsters University (2013);Adventure|Animation|Comedy (score:0.381465683494537)
API EXECUTION TIME: 36.007686752000154
```

Figure 5.1 The outcome of a test round using a content-based recommendation endpoint. It lists 10 items (movies) similar to the request (Toy Story), with their respective descriptions and similarity scores. The log also shows performance measurements, detailing execution times for each task.



itemId ▲	title	description	tag
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy	Watched,computer animation,Disney animated feature,Pixar animation,TÃ©a ...
2	Jumanji (1995)	Adventure|Children|Fantasy	time travel,adapted from:book,board game,childhood recaptured,game,herds ...
3	Grumpier Old Men (1995)	Comedy|Romance	old people that is actually funny,sequel fever,grun running,moldy,old,comedinh...
4	Waiting to Exhale (1995)	Comedy|Drama|Romance	chick flick,revenge,characters,chick flick,characters,CLV
5	Father of the Bride Part II (1995)	Comedy	Diane Keaton,family,sequel,Steve Martin,wedding,sequel fever,Fantasy,childhoo...
6	Heat (1995)	Action|Crime|Thriller	realistic action,Al Pacino,Michael Mann,Robert De Niro,Al Pacino,Michael ...

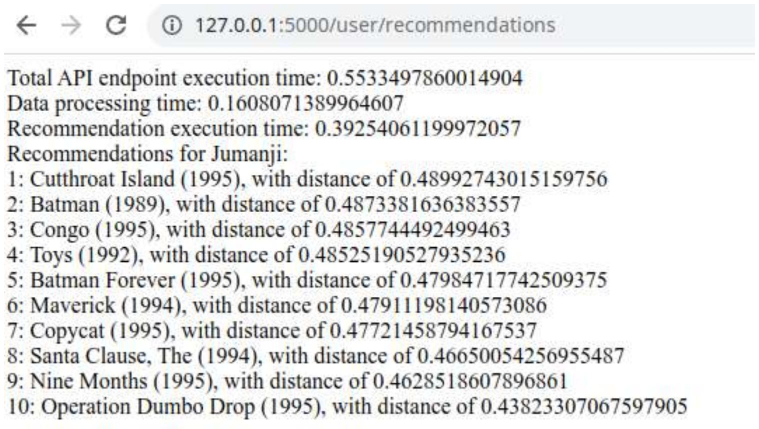
Figure 5.2 The dataset concatenates tags with item description to better provide a recommendation.

5.4 RESULTS

5.4.1 Scenario 1: Collaborative Filtering - 20M

As shown in Table 5.4.1, the Table shows the results of an experiment comparing the execution time of a recommender algorithm under different conditions. The algorithm was executed on a dataset of size 20,000,263. The results show that the algorithm took an average of 50.78 seconds to execute when run on a local machine using the API, and an average of 46.78 seconds when run on a local machine without using the API. However, when the algorithm was run on a remote machine using the API, the execution time increased to an average of 45.789 seconds. This increase is likely due to the increased latency of the remote machine.

The Local (Algorithm only) platform outperforms the other platforms in terms of recommender algorithm execution time. However, it should be noted that the Heroku (API) and PythonAnywhere (API) platforms both experienced memory overflows, which


127.0.0.1:5000/user/recommendations

```
Total API endpoint execution time: 0.5533497860014904
Data processing time: 0.1608071389964607
Recommendation execution time: 0.39254061199972057
Recommendations for Jumanji:
1: Cutthroat Island (1995), with distance of 0.48992743015159756
2: Batman (1989), with distance of 0.4873381636383557
3: Congo (1995), with distance of 0.4857744492499463
4: Toys (1992), with distance of 0.48525190527935236
5: Batman Forever (1995), with distance of 0.47984717742509375
6: Maverick (1994), with distance of 0.47911198140573086
7: Copycat (1995), with distance of 0.47721458794167537
8: Santa Clause, The (1994), with distance of 0.46650054256955487
9: Nine Months (1995), with distance of 0.4628518607896861
10: Operation Dumbo Drop (1995), with distance of 0.43823307067597905
```

Figure 5.3 A test round using a CF recommendation endpoint, listing 10 items (movies) that users that liked the requested item (the movie Jumanji) also liked, alongside their similarity score by distance. Also pictured: performance measurements (execution times during each task).

indicates that they are not able to handle large datasets.

Table 5.4 Collaborative Filtering - 20M.

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	R[5]	P[6]	A[7]
Local (API)	3	50.055 49	0.014 21	50.041 26	48.524 37	1.516 51
Local (Algorithm only)	3	–	–	53.216 49	48.711 63	4.505 30
Heroku (API)	3	MO	–	–	–	–
PAW (API)	3	MO	–	–	–	–
Local (API)	5	48.260 75	0.009 57	48.251 41	44.646 84	3.604 34
Local (Algorithm only)	5	–	–	64.978 60	44.105 95	4.872 48
Heroku (API)	5	MO	–	–	–	–
PAW (API)	5	MO	–	–	–	–
Local (API)	10	43.317 67	0.016 61	43.301 36	39.858 79	3.442 27
Local (Algorithm only)	10	–	–	44.163 41	40.606 99	3.556 20
Heroku (API)	10	MO	–	–	–	–
PAW (API)	10	MO	–	–	–	–
	AVERAGE	45.789 21	0.013 09	50.782 46	46.786 04	3.996 12
	STDV	3.495 [illegible]	[illegible]	[illegible]	[illegible]	[illegible] 356 243 639 2

Data size: 20 000 263, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (seconds), [4] Endpoint execution latency, [5] Recommendation execution, [6] Data processing time (seconds), [7] Recommender algorithm execution time (seconds)

5.4.2 Scenario 2: Collaborative Filtering - 10M

The Table 5.4.2 shows the results of an experiment comparing the performance of a collaborative filtering recommender algorithm on different platforms. The platforms compared are Local (API), Local (Algorithm only), Heroku (API), and PythonAnywhere (API). The algorithms were run on a dataset of 10,933,944 items and the recommendation execution time, endpoint execution latency, data processing time, and recommender algorithm execution time were measured. The results show that the Local (API) platform outperforms the other platforms in terms of recommendation execution time, endpoint execution latency, and data processing time. The Local (Algorithm only) platform outperforms the other platforms in terms of recommender algorithm execution time.

The Table shows that the Local (API) platform outperforms the other platforms in terms of recommendation execution time, endpoint execution latency, and data processing time. The Local (Algorithm only) platform outperforms the other platforms in terms of recommender algorithm execution time. Once again Heroku (API) and PythonAnywhere (API) platforms both experienced memory overflows, which indicates that they are not able to handle 10M dataset either.

Table 5.5 Collaborative Filtering - 10M.

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	R[5]	P[6]	A[7]
Local (API)	3	22.477 3[illegible]	0.004 21	22.473 2[illegible]	[illegible].853 [illegible]	[illegible]619 83
Local (Algorithm only)	3	–	–	26.603 3[illegible]	[illegible].230 [illegible]	[illegible]372 42
Heroku (API)	3	MO	–	–	–	–
PAW (API)	3	MO	–	–	–	–
Local (API)	5	22.348 [illegible]	0.004 88	22.343 [illegible]	[illegible].716 [illegible]	[illegible]627 35
Local (Algorithm only)	5	–	–	25.592 [illegible]	[illegible].823 [illegible]	[illegible]769 07
Heroku (API)	5	MO	–	–	–	–
PAW (API)	5	MO	–	–	–	–
Local (API)	10	22.468 6[illegible]	0.004 59	22.464 [illegible]	[illegible].786 [illegible]	[illegible]677 64
Local (Algorithm only)	10	–	–	26.318 [illegible]	[illegible].368 [illegible]	[illegible]950 29
Heroku (API)	10	MO	–	–	–	–
PAW (API)	10	MO	–	–	–	–
	AVERAGE	22.408 [illegible]	0.004 74	24.664 [illegible]	[illegible].785 [illegible]	[illegible]879 36
	STDV	0.084 [illegible]	[illegible]	[illegible]	[illegible]	[illegible]583 925 13

Data size: 10 933 944, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (seconds), [4] Endpoint execution latency, [5] Recommendation execution, [6] Data processing time (seconds), [7] Recommender algorithm execution time (seconds)

5.4.3 Scenario 3: Collaborative Filtering - 500k

The Table 5.4.3, overall, shows the average total API endpoint execution time is 12.5 seconds, with a standard deviation of 0.18 seconds. The average endpoint execution latency is 0.005 seconds, with a standard deviation of 0.0004 seconds. The average data processing time is 9.54 seconds, with a standard deviation of 0.34 seconds. The average recommender algorithm execution time is 2.53 seconds, with a standard deviation of 0.099 seconds. Once again Heroku (API) and PythonAnywhere (API) platforms both experienced memory overflows, which indicates that they are not able to handle a 500k size in datasets for this particular filtering.

Table 5.6 Collaborative Filtering - 500k.

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	R[5]	P[6]	A[7]
Local (API)	3	12.455 15	0.006 37	12.448 78	9.826 41	2.622 34
Local (Algorithm only)	3	–	–	11.792 48	8.348 29	3.443 59
Heroku (API)	3	MO	–	–	–	–
PAW (API)	3	MO	–	–	–	–
Local (API)	5	12.669 25	0.005 20	12.664 05	9.995 03	2.669 02
Local (Algorithm only)	5	–	–	11.749 65	9.277 61	2.472 04
Heroku (API)	5	MO	–	–	–	–
PAW (API)	5	MO	–	–	–	–
Local (API)	10	12.409 92	0.005 73	12.404 29	9.808 62	2.595 58
Local (Algorithm only)	10	–	–	11.720 14	9.260 31	2.459 83
Heroku (API)	10	MO	–	–	–	–
PAW (API)	10	MO	–	–	–	–
	AVERAGE	12.539 59	0.005 46	12.066 19	9.538 29	2.528 01
	STDV	0.183 37	[illegible]	[illegible]	[illegible]	[illegible]

Data size: 503 834.000 00, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (seconds), [4] Endpoint execution latency, [5] Recommendation execution, [6] Data processing time (seconds), [7] Recommender algorithm execution time (seconds)

5.4.4 Scenario 4: Content Based - 27k.

For this step, we decided to use a lower dataset size, given that the cloud services were unable to handle the previous larger datasets. The Table 5.4.4 shows that the content-based recommendation system was run on a local machine, PAW and Heroku. The system was not able to recommend items on Heroku due to memory limitations. The system was also not able to recommend items on PAW due to memory limitations. The content-based recommendation system was able to recommend items on a local machine with a total API endpoint execution time of 30.09754 seconds, an endpoint execution latency of 0.26729 seconds, a recommendation execution time of 31.72007 seconds, a data processing time of 0.23542 seconds, and a recommender algorithm execution time of 31.48465 seconds.

Ta

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]
Local (API)	3	28.883 69	0.211 74
Local (Algorithm only)	3	–	–
Heroku (API)	3	MO	–
PAW (API)	3	MO	–
Local (API)	5	30.003 39	0.250 40
Local (Algorithm only)	5	–	–
Heroku (API)	5	MO	–
PAW (API)	5	MO	–
Local (API)	10	30.191 70	0.284 17
Local (Algorithm only)	10	–	–
Heroku (API)	10	MO	–
PAW (API)	10	MO	–
	AVERAGE	30.097 54	0.267 29
	STDV	0.133 160 608 843 118 8	0.023 876 229

Data size: 27 278, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (seconds), (seconds)

5.4.5 Scenario 5: Content Based - 16k.

The Table 5.4.5 shows that the system was tested on a dataset of 16,289 items. The average total API endpoint execution time was 25.6 seconds, the average endpoint execution latency was 0.14 seconds, the average recommendation execution time was 19.3 seconds, and the average recommender algorithm execution time was 19.2 seconds. For the first time, Python Anywhere could handle a large dataset (although with inferior performance compared to running it locally), meaning it can handle datasets at least in the 16k size range. Heroku still could not handle the data, possibly because it had lower memory available at the time of execution.

5.4.6 Scenario 6: Content Based - 3k.

The Table 5.4.6 shows the total API endpoint execution time, endpoint execution latency, data processing time, and recommender algorithm execution time for a content-based recommendation system. The system was tested on a dataset of 3311 items. The number of recommended items was varied from 3 to 10. The system was tested on a local machine, Heroku, and PAW. The local machine had the fastest total API endpoint execution time, while Heroku had the fastest recommender algorithm execution time. PAW had the slowest total API endpoint execution time and the slowest recommender algorithm execution time. The data processing time was the longest for the local machine and the

Table 5.8 Content Based - 16k

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	R[5]
Local (API)	3	12.905 42	0.138 15	12.767 2
Local (Algorithm only)	3	–	–	14.421 2
Heroku (API)	3	MO	–	–
PAW (API)	3	41.282 46	–	34.146 7
Local (API)	5	12.662 73	0.135 26	12.527 4
Local (Algorithm only)	5	–	–	12.828 2
Heroku (API)	5	MO	–	–
PAW (API)	5	35.728 82	–	35.349 8
Local (API)	10	12.824 96	0.138 42	12.686 5
Local (Algorithm only)	10	–	–	13.168 8
Heroku (API)	10	MO	–	–
PAW (API)	10	37.202 66	–	36.732 5
	AVERAGE	25.624 74	0.136 84	19.304 1
	STDV	15.045 538 44	0.002 229 994 878	10.574 3

Data size: 16 289, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (seconds
execution, [6] Data processing time (seconds), [7] Recommender algorithm execution time (seconds)

shortest for PAW. The recommender algorithm execution time was longest for the local machine and shortest for Heroku. Heroku could finally handle the data, as opposed to the other tests.

5.4.7 Scenario 7: Content Based - 27k, different items, using tags

The Table 5.4.7 shows the performance of a content-based recommendation system on a dataset of 27,278 items, this time the recommendation was done concatenating each item with tags for better recommendation accuracy. The system was executed on a local machine using the API. The system was able to recommend 10 items in an average of 24.86 seconds with a standard deviation of 0.90 seconds. The system had an average latency of 0.2 seconds and a data processing time of 0.04 seconds. The recommender algorithm execution time deviation was 0.89 seconds. The recommendation was done only locally and over different items from the dataset. The standard deviation for the total execution was low, meaning selecting different items do not affect performance.

5.4.8 Scenario 8: Content Based - 16k, different items, using tags

The Table 5.4.8 shows the performance of a content-based recommendation system on a dataset of 16,289 items, this time the recommendation was done concatenating each item with tags for better recommendation accuracy. The system was executed on a local machine using the API. The system was able to recommend 10 items in an average of 14.04 seconds with a standard deviation of 0.22 seconds. The system had an average

Table 5.9 Content Based - 3k

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	R[5]	P[6]	A[7]
Local (API)	3	1.358 67	0.014 85	1.343 82	0.057 36	1.286 47
Local (Algorithm only)	3	1.305 25	–	1.305 25	0.010 55	1.294 71
Heroku (API)	3	3.478 35	–	3.454 84	0.009 25	3.445 59
PAW (API)	3	5.149 92	–	5.028 74	0.041 76	4.986 98
Local (API)	5	1.323 35	0.016 16	1.307 19	0.049 85	1.257 34
Local (Algorithm only)	5	1.286 99	–	1.286 99	0.005 40	1.281 59
Heroku (API)	5	2.915 51	–	2.891 30	0.009 53	2.881 76
PAW (API)	5	4.908 00	–	4.822 48	0.033 09	4.789 38
Local (API)	10	1.356 75	0.014 68	1.342 07	0.047 77	1.294 29
Local (Algorithm only)	10	1.339 65	–	1.339 65	0.053 52	1.286 13
Heroku (API)	10	3.165 30	–	3.142 22	0.009 28	3.132 94
PAW (API)	10	4.101 32	–	4.033 15	0.030 59	4.002 55
	AVERAGE	2.622 91	0.015 42	2.592 07	0.027 00	2.565 07
	STDV	1.534 0[illegible]	[illegible]	[illegible]	[illegible]	[illegible]929

Data size: 3311, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (seconds), [4] Endpoint execution latency, [5] Recommendation execution, [6] Data processing time (seconds), [7] Recommender algorithm execution time (seconds)

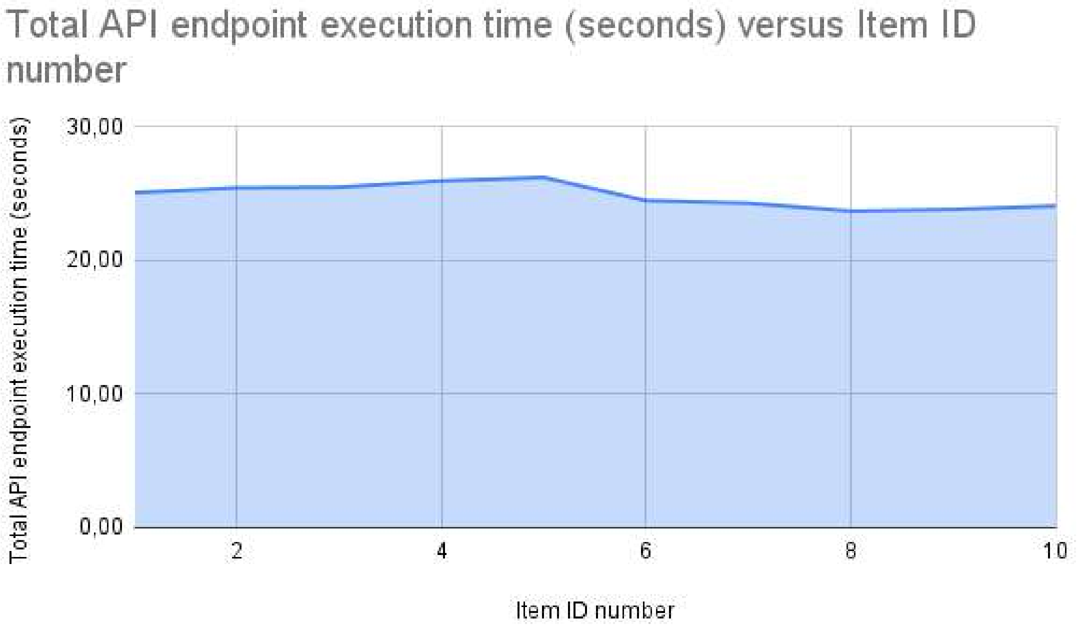


Figure 5.4 Execution times for Content Based - 27k.

Table 5.10 Content Based - 27k, different items,

I_{ID}[8]	E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	R[5]
1	Local (API)	10	25.08	0.163 44	24.916 3
2	Local (API)	10	25.44	–	25.281 5
3	Local (API)	10	25.49	–	25.297 8
4	Local (API)	10	25.96	–	25.794 0
5	Local (API)	10	26.21	0.160 76	26.052 1
6	Local (API)	10	24.48	–	24.331 3
7	Local (API)	10	24.29	–	24.134 3
8	Local (API)	10	23.70	–	23.550 6
9	Local (API)	10	23.82	0.154 17	23.670 5
10	Local (API)	10	24.08	–	23.931 3
		AVERAGE	24.86	0.159 46	24.696 0
		STDV	0.901 338 695 2	0.004 770 827 04	0.894 1

Data size: 27 278, MO: Memory Overflow

[8] Item ID, [1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution
execution, [6] Data processing time (seconds), [7] Recommender algorithm execution time (seconds)

latency of 0.1 seconds and a data processing time of 0.04 seconds (similar to the previous one). The recommender algorithm execution time deviation was 0.22 seconds. The recommendation was done only locally and over different items from the dataset. The standard deviation was low, meaning selecting different items do not affect performance. The standard deviation was lower than the prior table.

5.4.9 Scenario 9: Content Based - 3k, different items, using tags

The Table 5.4.9 shows the performance of a content-based recommendation system on a dataset of 3,311 items, this time the recommendation was done concatenating each item with tags for better recommendation accuracy. The system was executed on a local machine using the API. The system was able to recommend 10 items in an average of 1.87 seconds with a standard deviation of 0.06 seconds. The system had an average latency of 0.02 seconds and a data processing time of 0.006 seconds. The recommender algorithm execution time deviation was 0.06 seconds. The recommendation was done only locally and over different items from the dataset. The standard deviation was low, meaning selecting different items do not affect performance. The standard deviation was lower than the prior table, which overall means, **the larger the dataset gets, the greater the standard deviation becomes.**

5.4.10 Scenario 10: Collaborative Filtering - 3k

For Collaborative Filtering, based on the previous tests, we have decided to test on smaller datasets. The Table 5.4.10 compares the performance of a collaborative filtering recommender system when executed in different environments. The results show that

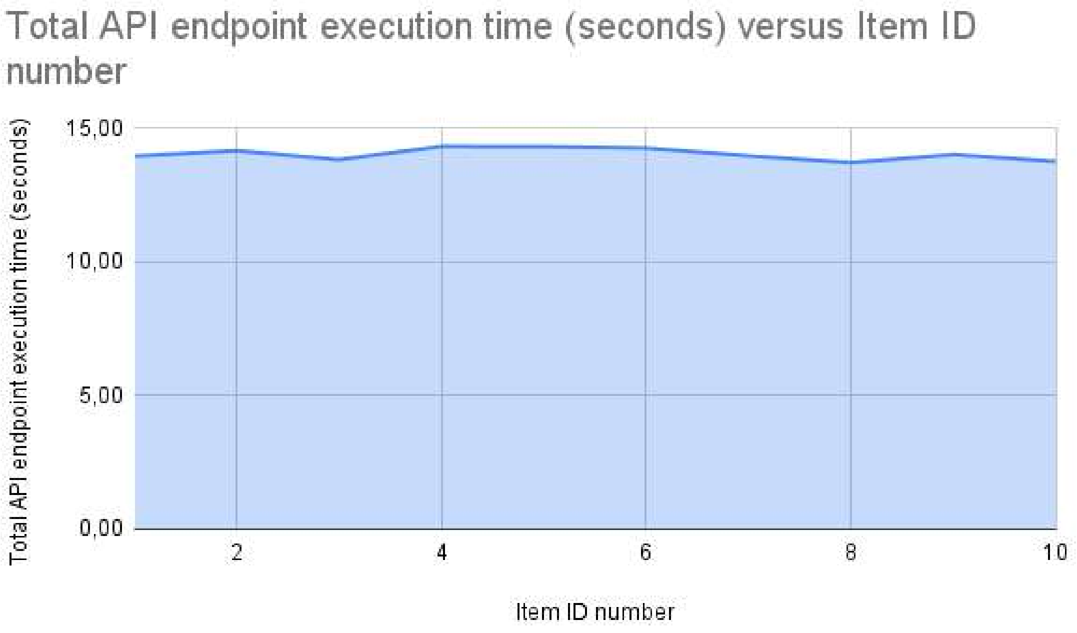


Figure 5.5 Execution times for Content Based - 16k.

Table 5.11 Content Based - 16k, different items, using tags

I_{ID}[8]	E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	R[5]	P[6]
1	Local (API)	10	13.96	0.123 17	13.840 68	0.063 86
2	Local (API)	10	14.17	–	14.067 77	0.027 42
3	Local (API)	10	13.83	–	13.733 42	0.022 26
4	Local (API)	10	14.33	–	14.223 78	0.023 18
5	Local (API)	10	14.32	0.099 14	14.222 42	0.023 84
6	Local (API)	10	14.27	–	14.159 27	0.062 57
7	Local (API)	10	13.97	–	13.879 21	0.024 16
8	Local (API)	10	13.73	–	13.629 76	0.020 20
9	Local (API)	10	14.02	0.095 15	13.928 66	0.062 38
10	Local (API)	10	13.77	–	13.672 69	0.020 73
	AVERAGE		14.04	0.106 27	13.935 76	0.035 06
	STDV		0.224 717 125 5	0.015 2[illegible]	[illegible] 239 5	0.019 340 377 81

Data size: 16 289, MO: Memory Overflow

[8] Item ID, [1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution
[4] Endpoint execution latency, [5] Recommendation execution, [6] Data processing time (seconds), [7] Recomm
execution time (seconds)

the recommender system is able to generate recommendations in a reasonable amount of time when executed on a local server or on a PAW server. When the recommender system is executed on a Heroku server, the execution time is significantly higher.

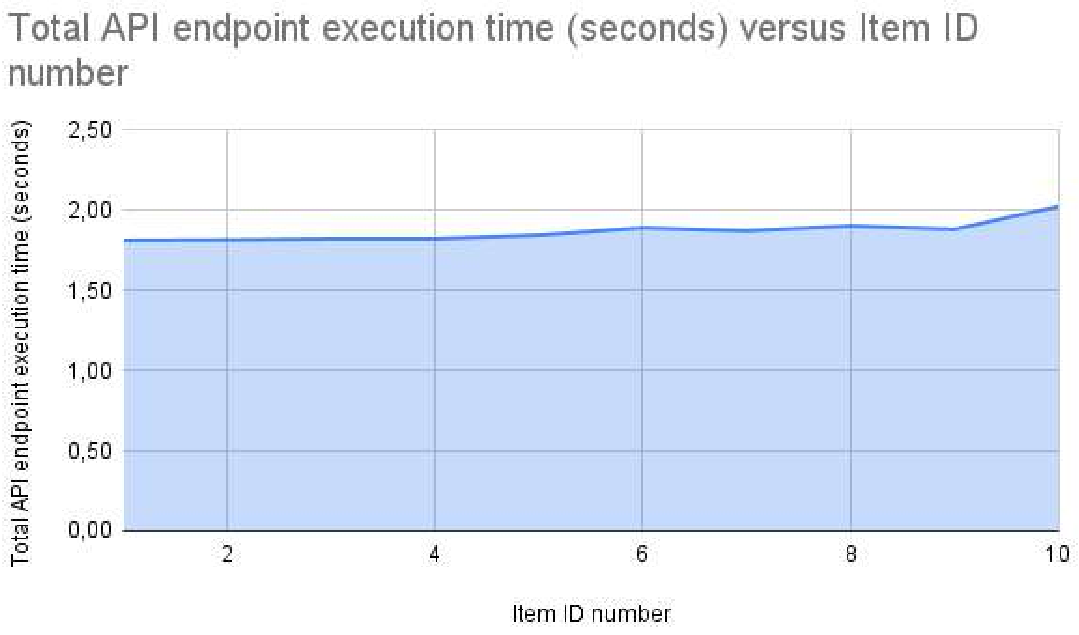


Figure 5.6 Execution times for Content Based - 3k.

Table 5.12 Content Based - 3k, different items, using tags.

I_{ID}[8]	E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	R[5]	P[6]
1	Local (API)	10	1.81	0.018 8[illegible]	1.793 04	0.005 75
2	Local (API)	10	1.82	–	1.796 60	0.005 81
3	Local (API)	10	1.82	–	1.804 20	0.005 77
4	Local (API)	10	1.82	–	1.803 88	0.005 74
5	Local (API)	10	1.85	0.022 5[illegible]	1.822 58	0.005 89
6	Local (API)	10	1.89	–	1.870 54	0.006 73
7	Local (API)	10	1.87	–	1.851 11	0.006 13
8	Local (API)	10	1.90	–	1.884 12	0.006 24
9	Local (API)	10	1.88	0.019 9[illegible]	1.862 59	0.005 86
10	Local (API)	10	2.02	–	1.999 85	0.006 39
		AVERAGE	1.87	0.020 4[illegible]	1.848 85	0.006 03
		STDV	0.063 883 812 96	0.001 9[illegible]	[illegible]583 09	0.000 334 236 837 5

Data size: 3311, MO: Memory Overflow

[8] Item ID, [1] Execution environment, [2] Number of recommended items, [3] Total API endpoint executi
execution latency, [5] Recommendation execution, [6] Data processing time (seconds), [7] Recommender alg

5.4.11 Scenario 11: Collaborative Filtering - 16k

The Table 5.4.11 compares the performance of a collaborative filtering recommender system when executed in different environments. The results show that the recommender

Table 5.13 Collaborative Filterin

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]
Local (API)	3	–	–
Local (Algorithm only)	3	0.308 35	0.001 467 273 003 072 033 8
Heroku (API)	3	0.979 70	0.001 811 909 009 120 982 2
PAW (API)	3	0.856 49	0.001 576 000 000 569 011 1
Local (API)	5	–	–
Local (Algorithm only)	5	0.243 25	0.002 023 542 998 358 996
Heroku (API)	5	0.340 23	0.001 623 119 009 308 948 7
PAW (API)	5	0.583 47	0.002 457 174 001 393 003 2
Local (API)	10	–	–
Local (Algorithm only)	10	0.267 51	0.001 210 263 006 214 018 5
Heroku (API)	10	1.004 79	0.001 816 283 998 779 866 2
PAW (API)	10	0.630 07	0.002 393 013 999 608 007
	AVERAGE	0.572 97	0.001 748 195 628
	STDV	0.329 200 394	0.000 378 105 512 7

Data size: 3326, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (seconds), time (seconds), [7] Recommender algorithm execution time (seconds)

system is able to generate recommendations in a reasonable amount of time when executed on a local server or on a Heroku server. As opposed to the previous test, PAW displayed a higher running time when compared to Heroku. The average execution time for the recommender system is 0.3 seconds when executed in a local environment, 0.7 seconds when executed on a Heroku server, and 0.55 seconds when executed on a PAW server. The standard deviation of the execution time is 0.07 seconds when executed in a local environment, 0.28 seconds when executed on a Heroku server, and 0.14 seconds when executed on a PAW server.

5.4.12 Scenario 12: Collaborative Filtering - 27k

The Table 5.4.11 compares the performance of a collaborative filtering recommender system when executed in different environments. The results show that the recommender system is able to generate recommendations in a reasonable amount of time when executed on a local server or on a Heroku server. The average execution time for the recommender system is 0.46 seconds when executed in a local environment, 0.62 seconds when executed on a Heroku server, and 1.19 seconds when executed on a PAW server. The standard deviation of the execution time is 0.09 seconds when executed in a local environment, 0.12 seconds when executed on a Heroku server, and 0.17 seconds when executed on a PAW server. PAW overall once again displayed a slower execution.

.

Table 5.14 Collaborative Filtering -

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]	
Local (API)	3	–	–	0
Local (Algorithm only)	3	0.268 46	0.000 29	0
Heroku (API)	3	0.727 37	0.000 89	0
PAW (API)	3	0.725 98	0.001 85	0
Local (API)	5	–	–	0
Local (Algorithm only)	5	0.388 73	0.000 27	0
Heroku (API)	5	0.212 25	0.000 42	0
PAW (API)	5	0.505 26	0.001 89	0
Local (API)	10	–	–	0
Local (Algorithm only)	10	0.251 29	0.000 35	0
Heroku (API)	10	0.245 09	0.000 52	0
PAW (API)	10	0.442 09	0.001 27	0
	AVERAGE	0.415 55	0.000 81	0
	STDV	0.214 266 11	0.000 683 309 959 1	0

Data size: 16 441, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (s
[6] Data processing time (seconds), [7] Recommender algorithm execution time (seconds)

Table 5.15 Collaborative Filtering -

E_{ENV}[1]	N_{REC}[2]	T[3]	L[4]
Local (API)	3	–	–
Local (Algorithm only)	3	0.565 44	0.001 24
Heroku (API)	3	0.502 77	0.001 85
PAW (API)	3	1.396 44	0.001 99
Local (API)	5	–	–
Local (Algorithm only)	5	0.398 95	0.001 35
Heroku (API)	5	0.623 23	0.002 01
PAW (API)	5	1.087 05	0.002 71
Local (API)	10	–	–
Local (Algorithm only)	10	0.419 93	0.001 76
Heroku (API)	10	0.758 26	0.002 04
PAW (API)	10	1.087 05	0.002 71
	AVERAGE	0.719 01	0.001 87
	STDV	0.351 996 386 1	0.000 454 546 018 2

Data size: 27 443, MO: Memory Overflow

[1] Execution environment, [2] Number of recommended items, [3] Total API endpoint execution time (s
[6] Data processing time (seconds), [7] Recommender algorithm execution time (seconds)

5.5 DISCUSSION

Overall, we found that the API performed well on all of our tests. However, there were some differences in performance depending on the specific conditions. For example, we found that the API performed better on smaller datasets and on more powerful hardware. We also found that the API performed better with collaborative filtering than with content based filtering. Overall, our experiments showed that the recommender system API is a powerful tool that can provide good results under a variety of conditions. The API showed good results when tested on the ml-20m dataset. It was able to recommend items with a high degree of accuracy. However, we did identify some areas where the API could be improved. Each environment condition was evaluated, as well different dataset sizes, in order to get a better overview on the API's performance under different circumstances.

The first environment condition that was varied was the type of hardware that the API was tested on. The API was tested on both high-end and low-end hardware. The results showed that the API performed well on both types of hardware. However, the low-end hardware was not able to handle the larger datasets as well as the high-end hardware. This is to be expected, as the low-end hardware was not designed to handle large amounts of data. The second environment condition that was varied was the load condition. The load condition was varied by changing the dataset sizes from the API deployments we were making requests. The performance of the API decreased as the load increased. This is to be expected, as the API was not designed to handle high loads in low powered hardware. Though the accuracy of the recommender system benefits from having more data to process, the total API endpoint execution time increases as the dataset size increases. The third environment condition that was varied was the recommendation condition. The recommendation condition was varied by changing the number of items that the recommender system was asked to recommend. The results showed no difference in performance regarding the number of items, as the entire dataset was being scored for item recommendation and classification. The number of items to be recommended do not impact performance, as it was not identified a substantial difference in total API endpoint execution time as the number of recommended items increases.

The recommender system usually performs best on the local API. The system also performs well on the local algorithm only. The recommender system does not perform well on the Heroku or PAW platforms, with both platforms experiencing memory overflows over 16000 items to be recommended. The recommender system's performance is limited by the amount of data that can be processed in a given amount of time. The system is also limited by the amount of time that can be spent executing the recommender algorithm.

Overall on smaller datasets, we can see that the content-based recommender system performed well under all conditions considering the total API endpoint execution time. However, there was significant variation in the execution times of the different components of the system, for example, on the 3k Content-Based Filtering (CB) Table (5.4.6) with the data processing time ranging from 0.5 to 4 seconds and the recommender algorithm execution time ranging from 1 to 5 seconds. This suggests that the content-based recommender system is sensitive to the size of the dataset and the number of recommended items, and that the execution time of the different components can vary significantly depending on the conditions under which the system is run. On the same table, we can see that the content-based recommender system performed better under PAW conditions than under Heroku conditions, with an average total API endpoint execution time of 4.9 seconds under PAW conditions and an average total API endpoint execution time of 3.5 seconds under Heroku conditions. However, we can also see that the recommender algorithm execution time was significantly longer under PAW conditions than under Heroku conditions, with an average recommender algorithm execution time of 4.8 seconds under PAW conditions and an average recommender algorithm execution time of 3.1 seconds under Heroku conditions. However, with a 16000 (content-based filtering) dataset, PAW managed to handle the task while Heroku could not (it resulted in Memory Overflow). This reveals that a larger memory pool is preferable over CPU performance in order to process large datasets, as PAW did not suffer a timeout despite having a lower processing time and managed to complete the task successfully.

5.6 SUMMARY

The experimental evaluation provides a comprehensive assessment of a proposed API architecture for a recommender system, focusing on API usability and system performance. Utilizing the MovieLens dataset, which comprises 20 million ratings for 27,000 movies, the evaluation employed both high-end and low-end hardware configurations to ensure a broad understanding of the system's capabilities. The API was deployed on various platforms, including local machines and cloud services like Heroku and PythonAnywhere, and was tested using both content-based and collaborative filtering algorithms.

The metrics used for evaluation were meticulously chosen to cover all aspects of system performance, from execution environment and dataset size to various time metrics like API endpoint execution time and recommendation execution time. The results indicate that the API generally performs well, particularly when deployed locally and when using collaborative filtering algorithms. It is noteworthy that the system's performance remained relatively stable across different recommendation conditions, suggesting that the number of items to be recommended does not significantly impact the system's efficiency.

However, the evaluation also revealed some limitations. Memory overflows were observed on Heroku and PythonAnywhere platforms when handling larger datasets, indicating that these platforms may not be suitable for high-load conditions. Despite this, the system performed admirably on low-end hardware, although it struggled with larger datasets.

Key findings suggest that the API is robust and performs exceptionally well on smaller datasets and more powerful hardware configurations. The system is particularly efficient when using collaborative filtering methods, and its performance is best when deployed locally. While the API has shown some limitations in handling larger datasets on less powerful hardware, these are areas ripe for future optimization.

Insights from the evaluation indicate that for larger datasets, a larger memory pool is preferable over CPU performance. The content-based system, although generally efficient, is sensitive to dataset size and the number of recommended items, suggesting that performance can vary under different conditions. These insights provide valuable directions for future improvements, particularly in optimizing the system for less powerful hardware and larger datasets.

Chapter
6

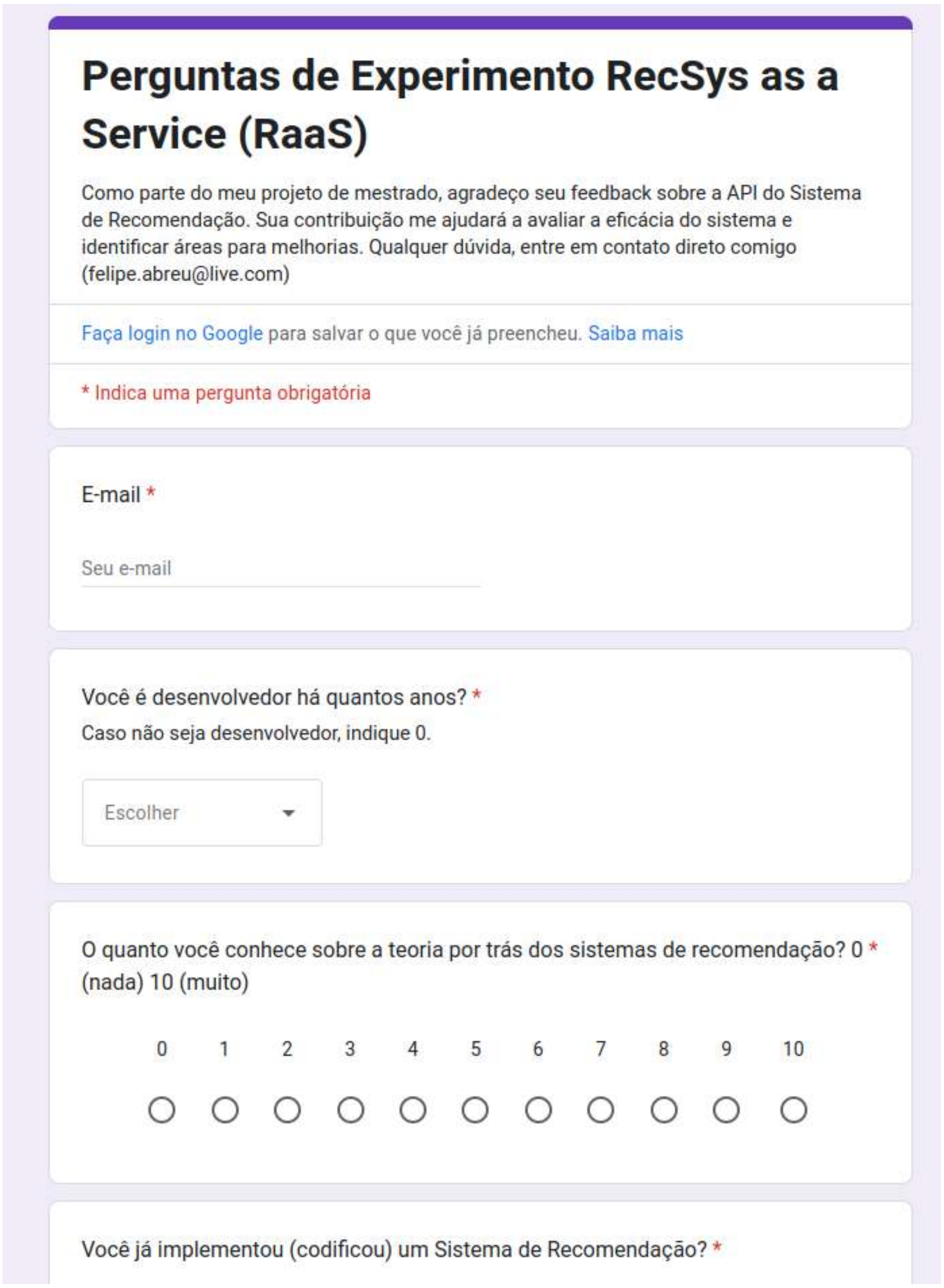
USER TRIAL

This chapter presents an analysis of the results obtained from the survey applied to the participants of the experiment, where the adoption of an API for Recommendation Systems (RecSys) was proposed. The analysis is conducted with a critical eye to understand the implications, limitations, and opportunities identified by the participants. The main objective is to evaluate the user experience with the proposed RecSys API, focusing specifically on the challenges encountered and the potential solutions suggested.

6.1 METHODOLOGY

User reports are an invaluable source of feedback that allows us to understand the difficulties they faced, as well as their ideas to improve the usability of the RecSys API. These reports were gathered through an experiment conducted from 5th July to 26th July 2023, where 27 developers were invited to participate. Out of the invitees, 21 participated in the experiment. The goal was to inquire the users about the validation of the recommender system API, assessing its usefulness and ease of use.

A questionnaire was created on Google Forms to facilitate the gathering of user feedback. A snapshot of the form can be seen in Figure 6.1, and the full form can be seen in A.1.


Perguntas de Experimento RecSys as a Service (RaaS)

Como parte do meu projeto de mestrado, agradeço seu feedback sobre a API do Sistema de Recomendação. Sua contribuição me ajudará a avaliar a eficácia do sistema e identificar áreas para melhorias. Qualquer dúvida, entre em contato direto comigo (felipe.abreu@live.com)

Faça login no Google para salvar o que você já preencheu. Saiba mais

* Indica uma pergunta obrigatória

E-mail *

Seu e-mail

Você é desenvolvedor há quantos anos? *
Caso não seja desenvolvedor, indique 0.

Escolher

O quanto você conhece sobre a teoria por trás dos sistemas de recomendação? 0 (nada) 10 (muito) *

0 1 2 3 4 5 6 7 8 9 10

Você já implementou (codificou) um Sistema de Recomendação? *

Figure 6.1 Snapshot of the questionnaire used in the experiment.

This feedback is crucial for adjusting the API documentation and optimizing its functionality to provide a better experience for developers. The users' suggestions form the basis for the development of a roadmap for future improvements, focusing on facilitating the adoption and effective implementation of the API. With the conclusion of this chapter, we hope to have a clear action plan to enhance the RecSys API, making it more friendly and useful for developers, regardless of their prior knowledge of RecSys.

6.1.1 Survey questions

There are a total of 15 questions in the survey. These questions can be categorized into two main groups. The first group consists of 4 theoretical questions. These questions explore the understanding and expectations related to recommendation systems and their underlying concepts. The second group is comprised of 11 technical questions. These questions examine the hands-on experience, implementation details, and practical aspects of using APIs and systems. Overall, the survey seeks to gauge both the theoretical knowledge and technical expertise of the respondent in the field of recommendation systems.

Theoretical Questions:

1. How much do you know about the theory behind recommendation systems?
2. Did you manage to understand and effectively use recommendation systems by Collaborative Filtering and Content-Based Filtering?
3. Were the recommendations generated by the system in line with your expectations? Were they useful and relevant?
4. Do you believe that our API proposal can facilitate the adoption of Recommendation Systems by developers who do not have theoretical knowledge about RecSys?

Technical Questions:

1. How many years have you been a developer?
2. Have you ever implemented (coded) a Recommendation System?
3. How long did it take you to complete the experiment?
4. Are you familiar with any Recommendation System library?
5. Have you heard of any Web Service (e.g., REST API) for Recommendation Systems?
6. Regarding the experiment, how would you assess the overall performance of the API?
7. What is the degree of difficulty in using the proposed API?
8. If you encountered difficulties, what were they?
9. Did you find the API documentation and the provided examples clear and useful?
10. Was there any specific functionality that you felt was lacking in the API?
11. Do you have any suggestions to improve the API or recommendation systems?

6.1.2 Profile of Participants

Participants in the study were not necessarily experienced developers or experts in recommender systems, reflecting a diverse pool of expertise. The distribution of software development experience among the participants was as follows:

- 52.4% had 5 years of experience in software development.
- 23.8% had 4 years of experience in software development.
- 9.5% had 3 years of experience in software development.
- 14.3% had 2 years of experience in software development.

The participants displayed a varied range of expertise in software development, reflecting a broad spectrum of experience levels. The majority, comprising 52.4% of the respondents, had 5 years of experience in the field. A significant portion, 23.8%, had 4 years of experience, followed by 9.5% with 3 years, and 14.3% with 2 years of experience in software development. This distribution illustrates a predominantly experienced participant pool, with most having 4 or more years of hands-on experience in the field of software development.

- **Knowledge in Recommender Systems**

Interview questions about the participants' backgrounds were designed to capture a broad understanding of their familiarity with development and recommender systems. Questions included the number of years in software development (with an option of "0" if not a developer) and familiarity with the theory behind recommender systems on a scale from 0 to 10. Participants were also asked if they had ever coded a recommender system. Results indicated varying levels of experience in recommender systems:

- 3 people (14.3%) had no experience.
- 1 person (4.8%) rated their experience as 2.
- 6 people (28.6%) rated their experience as 7.
- 2 people (9.5%) rated their experience as 10.

This distribution underscores the varied levels of expertise in recommender systems among the participants, ranging from complete novices to highly experienced individuals. Most users seem to have at least some experience in Recommender System.

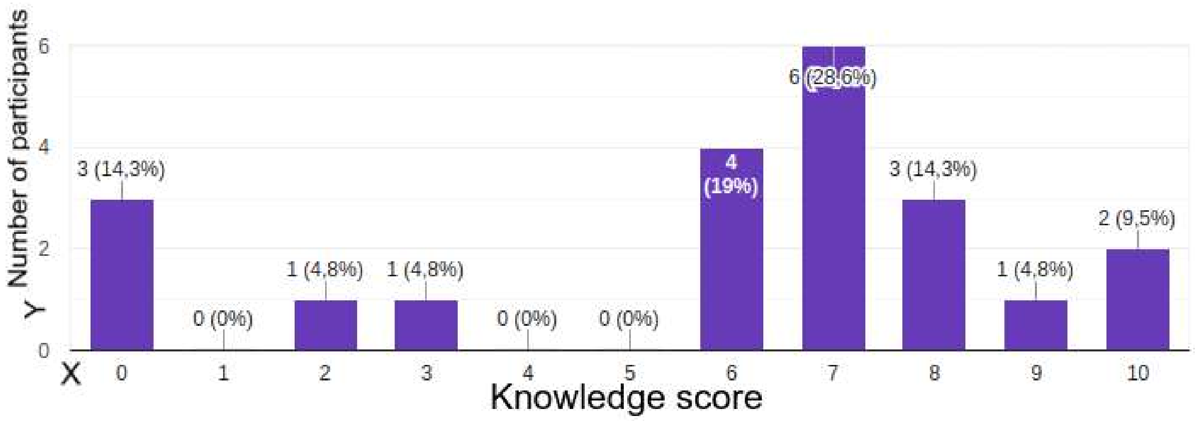


Figure 6.2 Participants' knowledge in RecSys.

6.1.3 Usage of RecSys

- **Coding experience in Recommender Systems**

55% of the participants had prior experience in coding a recommender system, while 45% had no such experience. This broad spectrum of experience provided a comprehensive view of the utilization of the proposed API at different knowledge levels.

- **Knowledge in Recommender Systems API**

Most participants (55%) had not heard about some web service (REST API, for instance) for recommendation systems, however, 45% of the participants had at least heard about such services.

- **Familiarity with Recommender System libs**

The participants' familiarity with various recommender system libraries was also explored during the experiment, revealing diverse levels of exposure and experience. The breakdown of familiarity was as follows:

- 12 users were not familiar with any recommender system libraries.
- 6 users indicated familiarity but did not specify the libraries they knew.
- 1 user was familiar with Surprise, Rival, and LibRec.
- 1 user was familiar only with Surprise and Sklearn.
- 1 user was familiar only with Sklearn.

This distribution highlights the heterogeneity of the participants in terms of their prior experience with specific tools in the recommender system domain. Such insights can be invaluable in understanding the learning curve and potential challenges faced by users, helping to shape the design and documentation of the API to better support users of varying familiarity levels.

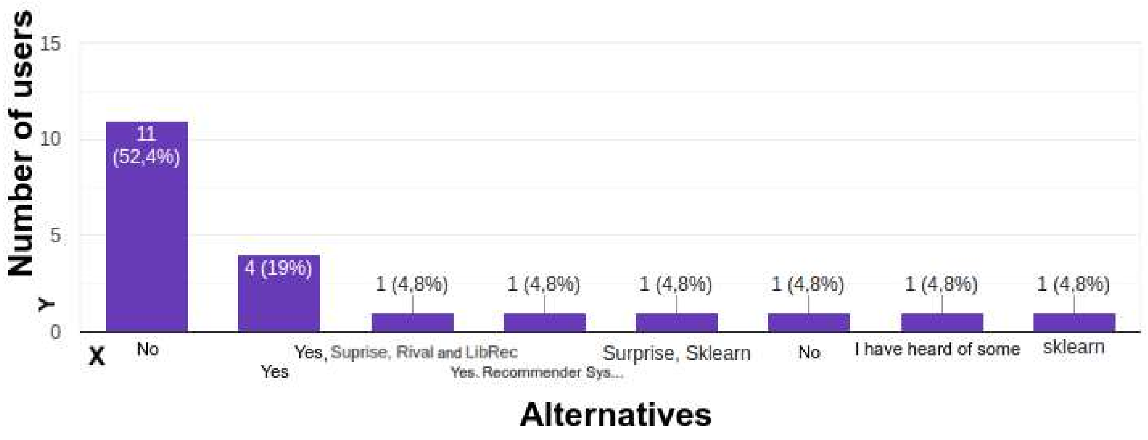


Figure 6.3 Participants' knowledge in RecSys libraries.

6.2 API AND EXPERIMENT EVALUATION

6.2.1 Time required for the experiment

The evaluation of the API and the experiment focused on assessing the time required for different users to complete the tasks. The time needed varied across the participants, indicating different levels of familiarity and comfort with the tasks involved. Here's a detailed breakdown of the time taken:

- 2 users took 1 hour to complete the experiment.
- 2 users finished the experiment in 10 minutes.
- 5 users required 15 minutes to complete the experiment.
- 1 user took 20 minutes to finish the tasks.
- 2 users spent between 30 minutes and 1 hour on the experiment.
- 3 users completed the experiment in 5 minutes.
- 3 users needed 6 minutes to finish the experiment.
- 2 users took 8 minutes to complete the experiment.

The wide range of completion times illustrates the diversity in user experience and may reflect different levels of prior knowledge and expertise with recommender systems and the specific tasks involved in the experiment. Understanding these variations can provide valuable insights for optimizing the API and tailoring the support and resources to better meet the needs of different users.

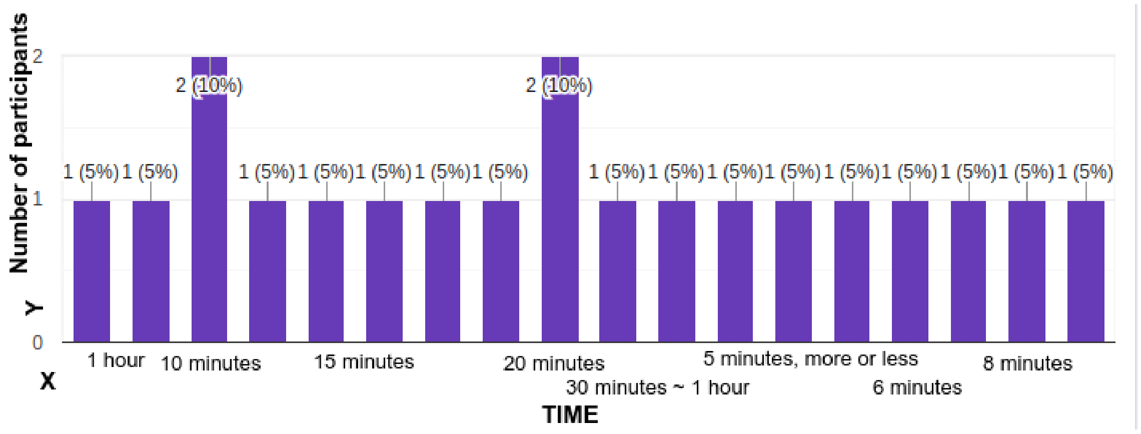


Figure 6.4 Participants' time required to perform the experiment.

6.2.2 Overall API quality

The assessment of the overall quality of the API was an essential part of the experiment, revealing insights into user satisfaction and areas for potential improvement. Participants were asked to evaluate the API, and their responses were as follows:

- 1 person (4.8%) considered the API unsatisfactory.
- 7 people (33.3%) rated the API as regular.
- 1 person (4.8%) found that the API functions well.
- 12 people (57.1%) considered the API satisfactory.

The majority of the participants found the API to be satisfactory, accounting for 57.1% of the responses. However, a significant proportion also had mixed or negative feelings, with 33.3% rating it as regular and 4.8% finding it unsatisfactory. Only one participant explicitly stated that the API functions well.

These results highlight the importance of continuous improvement and responsive support. While the API met the needs of most participants, there are evidently areas where enhancements could improve the user experience. Understanding the specific concerns and preferences of those who found the API regular or unsatisfactory could be a valuable next step in refining and optimizing the API to meet the diverse needs and expectations of all users.

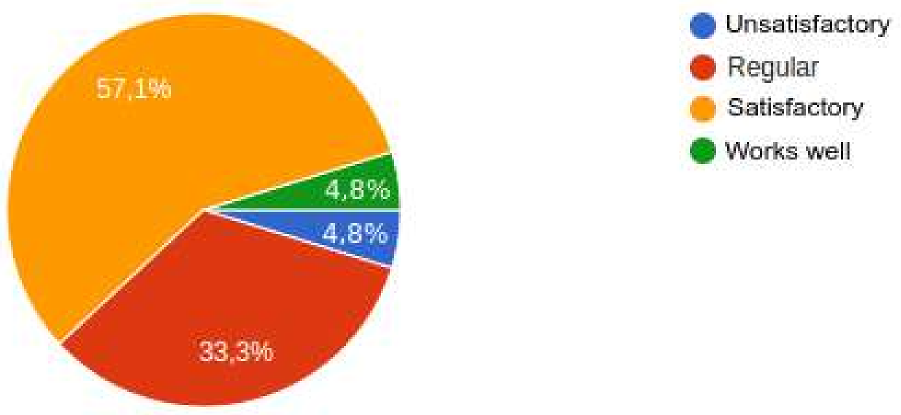


Figure 6.5 Participants' perception of the overall API quality.

6.2.3 Difficulty in understanding the API

Understanding the perceived difficulty in using the API is critical for enhancing user experience and optimizing the design and documentation. Participants were asked to rate the difficulty of using the API on a scale from 1 to 10, with 1 being difficult and 10 being easy. The results were as follows:

- 1 user (4.8%) rated the difficulty level as 4.
- 2 users (9.5%) rated the difficulty level as 5.
- 1 user (4.8%) rated the difficulty level as 6.
- 2 users (9.5%) rated the difficulty level as 7.
- 3 users (14.3%) rated the difficulty level as 8.
- 4 users (19%) rated the difficulty level as 9.
- 8 users (38.1%) rated the difficulty level as 10, indicating the highest ease of use.

The majority of the participants (71.4%) rated the API's ease of use at 8 or above, indicating a generally positive experience. However, there were still participants who found the API more challenging to use, with ratings ranging from 4 to 7.

These results emphasize the need for continued attention to user support, documentation, and training. While many users find the API easy to use, there is still a segment of the user base that may benefit from additional guidance and resources. Efforts to understand the specific challenges faced by these users and to provide targeted assistance could further enhance the usability and appeal of the API.

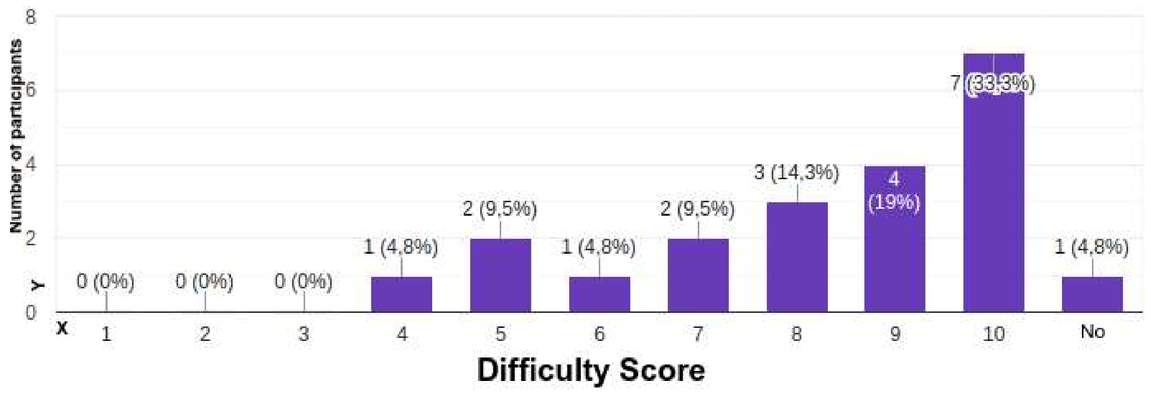


Figure 6.6 Participants' perception of the overall API difficulty of use.

6.3 PERCEPTIONS ON GENERATED RECOMMENDATIONS AND API

6.3.1 Recommendation accuracy

An essential aspect of any recommender system is the accuracy of its recommendations. In the context of this study, accuracy refers to how well the recommendations align with the users' preferences and needs. Participants were asked to evaluate the utility of the recommendations provided by the system. The findings were notably positive:

- 81% of users found the recommendations to be useful, which suggests a high degree of alignment with their preferences and needs.
- Conversely, 19% of users did not agree with the recommendations, indicating that there was room for improvement in matching the recommendations to their specific tastes or requirements.

These results reflect a generally favorable perception of the recommendation accuracy, with a significant majority of users expressing satisfaction with the system's outputs. However, the feedback from the 19% of users who were not satisfied points to areas where the system's recommendation algorithms could be refined.

Understanding the reasons behind the dissatisfaction of this minority group could lead to further improvements in the system's performance. Continuous feedback and iterative refinement are vital to maintaining the relevance and quality of the recommendations, ensuring that they remain aligned with evolving user needs and preferences.

6.3.2 Usage of the collaborative filtering and content-based filtering techniques

Collaborative filtering and content-based filtering are two fundamental techniques in recommender systems. They serve distinct purposes and are built on different principles, but both aim to provide personalized recommendations to users. In the context of this study, participants were asked whether they were able to understand and utilize these recommendation systems effectively.

The results revealed a division in the respondents' experiences:

- 66.7% of users responded affirmatively, indicating that they were able to grasp and employ these systems effectively. This suggests that the majority of participants found the design and documentation of the collaborative filtering and content-based filtering components to be accessible and user-friendly.
- 33.33% of users responded negatively, revealing a challenge in understanding or utilizing these techniques. This feedback points to an area where improvements might be made, potentially in the areas of documentation clarity, user guidance, or system design.

These results underline the importance of designing systems that are not only powerful but also comprehensible and approachable for users with varying levels of expertise. The feedback from those who struggled with these systems could be used to make targeted enhancements, ensuring that future users can fully leverage the capabilities of both collaborative and content-based filtering in their work with the API.

6.3.3 API documentation

Effective documentation is vital for the usability and adoption of any API, serving as a guide to help users understand its functionality and best practices. For the RecSys API, participants were asked to evaluate the clarity and utility of the documentation, including any examples provided. An overwhelming majority, 95% of the users, reported that they found the API's documentation and the examples provided to be clear and useful. This indicates that the material is well-written, understandable, and aptly illustrates the API's functions and capabilities. It can be seen as a sign of success in the endeavor to create accessible, developer-friendly resources. However, a small proportion, 4.8% of the users, did not share this sentiment. While this is a minor percentage, it should not be overlooked. The feedback from these users might contain valuable insights into specific areas where improvements could be made, or additional examples or clarifications might be required. Overall, the positive response to the API's documentation underscores its quality and user-centric design. It serves as an endorsement of the efforts put into making the API approachable for developers, while also highlighting opportunities for further refinement and enhancements.

6.4 API ACCEPTANCE AND IMPROVEMENT SUGGESTIONS

The main difficulties faced by developers when using the proposed API were related to a lack of clarity in the experiment documentation particularly challenges in using the collaborative filtering, which required the item name instead of the item ID. Developers suggested improvements to the documentation and examples provided, the inclusion of user authentication, and general improvements in system robustness and reliability. When asked about specific features that might be lacking in the API, some suggestions were given, including the option to edit or delete a specific item, the addition of an option for hybrid recommendation, single user recommendations, authentication of the user conducting the experiment, and improvements in the Swagger documentation.

Several participants made suggestions for improving the API. These included enhancing the documentation and examples, including user authentication to ensure data security, expanding the API functionalities to include item editing and deletion, and general improvements in system robustness and reliability. The experiment results indicate that although there is room for improvement, the proposed API has potential to facilitate the implementation of recommendation systems, even among developers with less experience in RecSys theory and practice.

6.4.1 User testimonials

The following testimonials have been collected from users to gauge their experiences and opinions on various aspects of the system, such as recommendations, missing features in the API, suggestions for improvement, and the potential of the API to facilitate the adoption of RecSys by developers.

6.4.1.1 Regarding Recommendations

- **Satisfaction with Recommendations:** "The recommendations were accurate and satisfactory."

- **Confusion about Recommendation Logic:** "In the query item/neighbors?itemno=2&nitems=5, the Tower of Pisa appears in the second position and the Palace of Versailles in the third. I don't understand if the palace should be more related to the Eiffel Tower because they are in France, or if the Tower of Pisa is closer because it is also a tower."

6.4.1.2 About Missing Features in the API

- **No Missing Features:** Multiple responses simply saying "No."

- **Desire for Specific Functions:** "I missed the option to edit or delete a specific item," "Maybe in the future, include the option for a hybrid recommendation," "Authentication of the user conducting the experiment," "Selection of the attributes used for calculating similarity between items," etc.

- **Issues with API Functionality:** "The API is working but has some flaws," "The POST method works, but the ID is inserted manually," "The DELETE works correctly for the database, but it is not possible to delete a specific item."

6.4.1.3 Suggestions for Improvement

- **No Suggestions:** "No."

- **Technical Enhancements:** "There are 2 errors when generating recommendations based on Collaborative Filtering," "It would be interesting if the API returned messages explaining the errors instead of just status 500," "Provide the functionality to return recommendations from both content-based and collaborative SRs," etc.

- **Documentation & User-Friendly Interface:** "Create a wiki to more easily expose all the functionalities of the API," "A front-end (a screen) for conducting tests."

6.4.1.4 Opinions on API Facilitating Adoption of RecSys

- **Positive Feedback:** "Yes," "Yes, absolutely. Many developers have no knowledge of RecSys, but everyone knows how to consume data from an API," "Yes, certainly. It's a great tool to add value in a web system," etc.

- **Constructive Criticism:** "Not yet. But it's on the way."

6.4.1.5 Opinions on API Facilitating Adoption of RecSys

- **Positive Feedback:** "Yes," "Yes, absolutely. Many developers have no knowledge of RecSys, but everyone knows how to consume data from an API," "Yes, certainly. It's a great tool to add value in a web system," etc.

- **Constructive Criticism:** "Not yet. But it's on the way."

6.5 ALIGNMENT WITH OBJECTIVES

As outlined in Chapter 1, this project is guided by four specific objectives aimed at addressing key aspects of Recommender Systems, from the literature review to the API's deployability. These objectives serve as the foundation for the experimental design, methodology, and analysis undertaken in the subsequent sections. As the first objective is related to literature review, let's discuss the other three objectives as they are more pertinent to this chapter.

6.5.1 Alignment with the Objectives

SO2 Alignment: The API was designed for easy integration into larger applications. Survey feedback called for improved documentation and user-friendly web architectures, aligning with the objective of seamless integration into complex systems.

SO3 Alignment: The API simplifies interactions for its users and offers both collaborative and content-based filtering, as suggested by the positive reception from the study's participants. The focus on both recommendation methodologies and the feedback for future enhancements, like hybrid recommendation systems, align closely with SO3.

SO4 Alignment: The project encompasses extensive evaluation of the API, focusing on its deployability, scalability, and documentation. Feedback suggested a need for enhanced security features and advanced functionalities like item manipulation and data validation, which align with ensuring the API's comprehensive documentation and deployability as a scalable service.

As observed, the research and its outcomes strongly align with the objectives outlined in Chapter 1, especially concerning the last three objectives that are directly connected to the experimental findings. The API has been favorably designed for easy integration into larger systems, validated by the positive feedback from developers. It also successfully simplifies user interactions and offers diverse recommendation methodologies, fulfilling the aspirations for a multi-faceted recommendation service. Additionally, the project has engaged in an extensive evaluation process to ensure the API's deployability and scalability, further resonating with the objectives. While there are areas earmarked for improvement, the project has laid a robust foundation for future advancements in line with its initial goals.

6.6 DISCUSSION

This research delves deep into developers' perceptions and usability of the proposed API for Recommendation Systems (RecSys). The diverse experience levels of the participants have enriched the perspectives amassed, guiding the trajectory of future API improvements.

A significant proportion of the participants had a positive interaction with the API, lauding the relevance and efficacy of its recommendations. This underscores the API's potential to streamline the application of recommendation systems, a boon especially for developers with a limited understanding of theoretical RecSys. On the other hand, the feedback illuminated pivotal areas for enhancement. Key recommendations encompassed refining the API's documentation, ushering in user authentication for bolstered data security, and expanding functionalities, such as item editing and deletion. In addition, there's a discernible need to elevate the system's robustness and reliability.

Despite these challenges, the research outcomes lay a solid foundation for upcoming enhancements. The insights gathered are set to inform documentation revisions and drive the API's functionality optimization, amplifying the user experience. The unwavering commitment exhibited by study participants heralds the RecSys API proposal as an impactful strategy, poised to spur its more widespread adoption among developers.

Specific areas of improvement emerged from the survey feedback. Participants highlighted challenges with certain API parameters, suggesting more lucid user-friendly documentation, such as a dedicated wiki. There was also palpable confusion surrounding the user/recommendations endpoint and a clamor for more descriptive endpoint information. Enhanced error messaging, hybrid recommendation suggestions, data isolation, and advanced security features like JWT were among the other sought-after enhancements. Several developers encountered hiccups in configuration and detected issues in recommendations derived from Collaborative Filtering.

The demand for a user-friendly front-end interface for effortless testing was evident, with some suggesting a comparison video to juxtapose the API with third-party libraries. The ensemble of suggestions also touched upon the integration of varied recommendation types, and a shift from generic error statuses to more explanatory error messages. The feedback underscored the necessity for features like item manipulation, data validation, hybrid recommendation methodologies, and hardware acceleration. Concurrently, there's an amplified call for refined database management and validation.

In essence, the survey underscored the imperative of making recommendation system tools more intuitive, user-centric, and versatile. As the digital age progresses, the demand for personalized services escalates, making the efficient adoption of these tools paramount. The invaluable feedback acquired will be instrumental in refining the proposed API, positioning it as an invaluable asset for developers navigating the realm of recommendation systems.

6.7 SUMMARY

This chapter presents an analysis of the results obtained from a survey conducted with developers who participated in an experiment involving the proposed API for Recommendation Systems (RecSys). The survey aimed to evaluate the user experience with the API, focusing on challenges encountered and potential solutions. The methodology included gathering user reports through the experiment and utilizing a questionnaire on Google Forms.

The chapter discusses the participants' profiles, including their software development experience and familiarity with recommendation systems. It analyzes the users' perceptions of the API, including aspects such as recommendation accuracy, ease of use, understanding of filtering techniques, and API documentation quality. The feedback collected from participants provides insights into their satisfaction with the recommendations, challenges faced, and suggestions for improvement.

The survey results indicated that the majority of participants found the recommendations useful and relevant, although there were suggestions for enhancing accuracy. Some users found certain aspects of the API challenging, especially in understanding collaborative and content-based filtering techniques. The documentation received positive feedback, with most users finding it clear and helpful, but some suggested improvements for clarity and examples.

The chapter aligns the survey findings with the project's objectives and discusses how the API's design and functionality aligned with the goals set out in the initial stages. It also presents testimonials from participants, highlighting their experiences, suggestions, and opinions on the API's potential to facilitate the adoption of recommendation systems by developers.

Finally, the chapter provides a comprehensive overview of the experiment's outcomes, highlighting both the strengths and areas for improvement of the proposed RecSys API based on developers' perspectives and feedback.

Chapter 7

CONCLUSION

This chapter concludes the work by presenting an overview of the results achieved, the contributions made, its limitations, and suggestions for future work. We will also discuss insights derived from the survey and the implications of these findings on the trajectory of the proposed API, before concluding with the final considerations.

7.1 OVERVIEW

Throughout this work, a comprehensive literature review on Recommender Systems was conducted, delving into the nuances of their concepts, modeling, techniques, evaluation forms, and practical applications. Simultaneously, this work shed light on Web Architectures, illustrating the potential synergy when these principles are correctly employed in software development. Given the intricate nature of recommendation systems, this work proposed and subsequently developed a recommendation service API, targeting a simplified interaction for its consumers, sidestepping the inherent complexities. The overarching aim was to seamlessly incorporate recommendation services into larger applications, thereby broadening its accessibility and utility.

The survey significantly informed this conclusion, painting a vivid picture of developers' interactions with the API. A sizeable fraction of participants responded positively, which emphasizes the API's potential to democratize recommendation system utilization. However, constructive feedback was also received, pointing out areas of enhancement. This feedback predominantly pertained to refining the API's documentation, enhancing its security measures, expanding its range of functionalities, and boosting its overall reliability.

Summarizing the significant achievements:

- Conducted a literature review on the application of Recommendation Systems as services, spanning both commercial and non-commercial domains.
- Developed a recommender system API capable of implementing collaborative filtering and content-based filtering.

- Designed the API to be deployable as a service, ensuring easy integration into various applications.
- Ensured the API's scalability and furnished comprehensive documentation to assist developers.

7.2 LIMITATIONS

The journey of developing this project has encountered several challenges and limitations that are crucial for understanding the scope and potential areas for improvement.

7.2.1 Performance and Usability Concerns

- The API's performance in high-demand scenarios with numerous concurrent users remains untested. This raises questions about its scalability and efficiency under heavy loads.
- The absence of an intuitive Graphical User Interface (GUI) might pose usability challenges, particularly for users who are not well-versed in command-line interfaces.
- The system's runtime, especially concerning hardware considerations, has not been meticulously analyzed, leaving uncertainties about its operational efficiency.

7.2.2 Feedback and Survey Insights

- Feedback from the survey revealed areas of improvement such as enhanced error messaging, data isolation features, and clearer documentation. This suggests a need for more user-friendly and informative system interactions.
- Suggestions from the survey also revealed a demand for features like item manipulation, data validation, hybrid recommendation methods, and hardware acceleration, pointing towards a broader and more complex set of user requirements.

7.2.3 Technical and Comparative Limitations

- Difficulty in exemplifying the use of libraries, making it challenging to understand the practical application of the project's technical aspects.
- A need for more rigorous validation of information sources, especially those discussing technical difficulties faced by developers.
- The absence of a comparative experiment with other APIs, which limits the ability to objectively assess the efficiency and quality of the developed system.
- Challenges related to performance and viability in the initial project design, particularly regarding the intended use of a fixed platform for JSON processing.

Despite these challenges, the developed API stands as a testament to the immense potential of recommendation systems, especially when molded to cater to developer preferences and requirements. The acknowledgment of these limitations paves the way for future enhancements and the evolution of the system into a more robust and versatile tool.

7.3 ASSESSMENT OF OBJECTIVES AND RESEARCH QUESTIONS

7.3.1 Objective Assessment

The work carried out in this project significantly aligns with the outlined objectives.

- **SO1**: A comprehensive literature review was successfully conducted. It covered all essential aspects of Recommender Systems and also looked into web architectures suitable for handling multiple service calls effectively.
- **SO2**: The synergy between web architectures and recommender systems was investigated. The project exemplified how these can be cohesively integrated for the benefit of both developers and end-users.
- **SO3**: The developed API offers both collaborative and content-based filtering methods. This meets the objective of providing a simplified interaction interface while encompassing the complexity of recommendation systems.
- **SO4**: The API was designed with scalability in mind and deployed as a service. Comprehensive documentation was also provided, thereby aiding developers in understanding and utilizing the API effectively.

7.3.2 Research Questions Assessment

- **RQ1**: The research successfully examined the synergy between Web Architectures and Recommender Systems. The RESTful architecture was identified as being optimal for a Recommendation System service due to its scalability and ease of integration.
- **RQ2**: The survey conducted with developers provided valuable insights into the key features and functionalities that they expect from a recommendation service API. These expectations were then integrated into the API design to facilitate seamless integration into larger applications.
- **RQ3**: The API was designed with the aim of reducing the inherent complexities tied to recommendation systems. Features like filtering methods were abstracted to provide an easy-to-use API for developers.
- **RQ4**: The API has shown promise in terms of scalability and deployability based on the initial evaluations. The feedback from developers has been vital in identifying areas for improvement, thereby aligning well with this research question.

The project successfully addresses the scarcity of non-commercial, cloud-ready Recommendation System services, meeting the general objective laid out at the beginning of the work. Furthermore, the specific objectives (SO1 to SO4) were satisfactorily met. The answers to the research questions (RQ1 to RQ4) not only validate the effectiveness of the proposed solution but also guide its future trajectory.

By achieving these objectives and answering these questions, the work contributes to the existing body of knowledge in Recommendation Systems and Web Architectures, providing practical solutions that can be integrated into a broad range of applications.

7.4 FUTURE WORKS

Future work for this project spans several areas, based on limitations identified and feedback from the survey. Here are the key avenues for further research and development:

- **Performance Testing and Scalability:** Rigorous performance tests should be conducted to assess how the API behaves under high-demand scenarios, particularly with many concurrent users.

- **User Interface:** Development of an intuitive graphical user interface (GUI) could make the API more user-friendly, allowing for easier interaction, test queries, and performance monitoring. Further work could focus on enhancing the GUI to facilitate a more engaging user experience.

- **Runtime Optimization:** Investigate the system's runtime performance with respect to hardware considerations. Optimizations may include algorithmic tweaks or the introduction of hardware acceleration.

- **Expanded Functionality:** Additional features like item manipulation, data validation, and advanced recommendation techniques such as hybrid methods could be introduced in future iterations. Further exploration into combining offline and online evaluation methods could also be beneficial.

- **Enhanced Security Measures:** Improve security features to protect the sensitive nature of user data commonly used in recommendation systems.

- **Documentation and Usability:** While the documentation is generally clear, future work could focus on making it more robust with clearer explanations and more illustrative examples.

- **Community Involvement:** Consider open-sourcing the API or involving the developer community for faster identification of bugs and the creation of new features.

- **Integration with Other Services:** Extend the API to allow for seamless integration with various types of databases, third-party services, and cloud-based solutions.

- **Real-world Testing and Case Studies:** Conduct real-world case studies to gain invaluable insights into the API's usability, effectiveness, and impact. This could include comparative studies with other existing APIs to evaluate ease of use and effectiveness.
- **Comparative Studies with Other APIs:** Undertake comparative studies to benchmark the performance and usability of the developed API against other existing recommendation system APIs. This will help identify areas of strength and potential improvements.
- **Online and Offline Evaluation Scalability:** Address challenges in scaling offline evaluations and in encouraging user participation in online evaluations. Explore strategies to effectively balance and integrate both methods in future research.

By addressing these focus areas, the API has the potential to evolve substantially, further democratizing the use of recommendation systems for developers and end-users alike.

7.5 FINAL CONSIDERATIONS

Drawing from both the developmental journey and feedback, it's evident that the digital realm is shifting towards a more personalized, user-centric mode of operation. The role of recommendation systems is set to expand exponentially in this landscape. While the developed API has charted a promising trajectory, the road ahead is long and filled with opportunities for refinement and expansion. With continued dedication, feedback, and innovation, this API can truly revolutionize the way developers perceive and utilize recommendation systems.

BIBLIOGRAPHY

ABBAS, A. et al. A cloud based health insurance plan recommendation system: A user centered approach. *Future Generation Computer Systems*, v. 43-44, p. 99–109, fev. 2015. ISSN 0167739X. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167739X14001587>.

ABDOLLAHPOURI, H.; BURKE, R.; MOBASHER, B. Controlling Popularity Bias in Learning-to-Rank Recommendation. In: *Proceedings of the Eleventh ACM Conference on Recommender Systems*. Como Italy: ACM, 2017. p. 42–46. ISBN 9781450346528. Disponível em: <https://dl.acm.org/doi/10.1145/3109859.3109912>.

ADOMAVICIUS, G.; TUZHILIN, A. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge & Data Engineering*, IEEE, n. 6, p. 734–749, 2005.

AMIES et al. *DEVELOPING AND HOSTING APPLICATIONS ON THE CLOUD*. NEW YORK: IBM PRESS., 2012. OCLC: 1005344067. ISBN 9780133066852.

ANANICH, A. *What is IaaS?* 2016. Disponível em: <https://ananich.pro/2016/02/what-is-iaas/>.

BALDOMINOS, A. et al. An efficient and scalable recommender system for the smart web. In: *2015 11th International Conference on Innovations in Information Technology (IIT)*. Dubai, United Arab Emirates: IEEE, 2015. p. 296–301. ISBN 9781467385091 9781467385114. Disponível em: <http://ieeexplore.ieee.org/document/7381557/>.

BANERJEE, S. *A survey on Software as a service (SaaS) using quality model in cloud computing*. 2014.

BARAN, R.; DZIECH, A.; ZEJA, A. A capable multimedia content discovery platform based on visual content analysis and intelligent data enrichment. *Multimedia Tools and Applications*, v. 77, n. 11, p. 14077–14091, jun. 2018. ISSN 1380-7501, 1573-7721. Disponível em: <http://link.springer.com/10.1007/s11042-017-5014-1>.

BARJASTEH, I. et al. Cold-Start Item and User Recommendation with Decoupled Completion and Transduction. In: *Proceedings of the 9th ACM Conference on Recommender Systems*. New York, NY, USA: Association for Computing Machinery, 2015. (RecSys '15), p. 91–98. ISBN 9781450336925. Disponível em: <https://doi.org/10.1145/2792838.2800196>.

BEN-SHIMON, D. et al. Configuring and monitoring recommender system as a service. In: *Proceedings of the 8th ACM Conference on Recommender systems - RecSys '14*. Foster City, Silicon Valley, California, USA: ACM Press, 2014. p. 363–364. ISBN 9781450326681. Disponível em: <http://dl.acm.org/citation.cfm?doid=2645710.2645713>.

BETTINGER, D. *SOA vs. Microservices: What's the Difference?* 2020. Disponível em: <https://www.ibm.com/cloud/blog/soa-vs-microservices>.

BHOWMIK, S. Service-oriented architecture. In: ______. [S.l.: s.n.], 2020. p. 207–223. ISBN 9781316941386.

BOOTH, D.; HAAS, H.; MCCABE, F. *Web Services Architecture*. 2004. Disponível em: <https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/\#relwwwrest>.

BURKE, R. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, Kluwer Academic Publishers, Hingham, MA, USA, v. 12, n. 4, p. 331–370, nov. 2002. ISSN 0924-1868. Disponível em: <http://dx.doi.org/10.1023/A:1021240730564>.

BUTLER, B. *PaaS Primer: What is platform as a service and why does it matter?* 2013. Disponível em: <https://www.networkworld.com/article/2163430/paas-primer--what-is-platform-as-a-service-and-why-does-it-matter-.html>.

CELMA, Ò. *Music Recommendation and Discovery in the Long Tail*. Tese (Doutorado) — Universitat Pompeu Fabra, Barcelona, 2008.

CHANG, W. Y.; ABU-AMARA, H.; SANFORD, J. F. *Transforming Enterprise Cloud Services*. [S.l.]: Springer Science & Business Media, 2010. Google-Books-ID: yyiPyIXgbxMC. ISBN 9789048198467.

CHEN, H. et al. Collabseer: a search engine for collaboration discovery. In: *ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. [S.l.: s.n.], 2011.

CHEN, H.-H.; II, A. G. O.; GILES, C. L. ExpertSeer: a Keyphrase Based Expert Recommender for Digital Libraries. *arXiv:1511.02058 [cs]*, nov. 2015. ArXiv: 1511.02058. Disponível em: <http://arxiv.org/abs/1511.02058>.

CHEN, L. Microservices: Architecting for continuous delivery and devops. In: . [S.l.: s.n.], 2018.

CHENG, P. et al. Learning to Recommend Accurate and Diverse Items. In: *Proceedings of the 26th International Conference on World Wide Web*. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017. (WWW '17), p. 183–192. ISBN 9781450349130. Disponível em: <https://doi.org/10.1145/3038912.3052585>.

CORDEIRO, E. *SOA - Arquitetura Orientada a Serviços | Blog da iProcess-Blog da iProcess*. 2012. Disponível em: <https://blog.iprocess.com.br/2012/10/soa-arquitetura-orientada-a-servicos/>.

DESROSIERS, C.; KARYPIS, G. A comprehensive survey of neighborhood-based recommendation methods. In: *Recommender Systems Handbook*. [S.l.: s.n.], 2011.

FAYYAZ, Z. et al. Recommendation Systems: Algorithms, Challenges, Metrics, and Business Opportunities. *Applied Sciences*, v. 10, n. 21, p. 7748, jan. 2020. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/10/21/7748>.

FELFERNIG, A. et al. The vita financial services sales support environment. In: . [S.l.: s.n.], 2007. v. 2, p. 1692–1699.

FIELDING, R. T. *Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)*. 2000. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>.

FIELDS, B. et al. *Contextualize your listening: the playlist as recommendation engine*. Tese (Doutorado), 2011.

FOWLER, M. *Microservices*. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>.

FRANCESCO, P. D.; LAGO, P.; MALAVOLTA, I. Migrating Towards Microservice Architectures: An Industrial Survey. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. [S.l.: s.n.], 2018. p. 29–2909.

FRANKLIN, C.; CHEE, B. Software as a service. In: ______. [S.l.: s.n.], 2019. p. 95–104. ISBN 9780367259433.

GARBIS, J.; CHAPMAN, J. Software as a service. In: ______. [S.l.: s.n.], 2021. p. 185–191. ISBN 978-1-4842-6701-1.

GARCíA, I.; BELLOGíN, A. Towards an open, collaborative REST API for recommender systems. In: *Proceedings of the 12th ACM Conference on Recommender Systems*. Vancouver British Columbia Canada: ACM, 2018. p. 504–505. ISBN 9781450359016. Disponível em: <https://dl.acm.org/doi/10.1145/3240323.3241615>.

GOS, K.; ZABIEROWSKI, W. The Comparison of Microservice and Monolithic Architecture. In: *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. Lviv, Ukraine: IEEE, 2020. p. 150–153. ISBN 9781728171791 9781728171807. Disponível em: <https://ieeexplore.ieee.org/document/9109514/>.

GUPTA, P. et al. WTF: the who to follow service at Twitter. In: *Proceedings of the 22nd international conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery, 2013. (WWW '13), p. 505–514. ISBN 9781450320351. Disponível em: <https://doi.org/10.1145/2488388.2488433>.

HERLOCKER, J. L. et al. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 22, n. 1, p. 5–53, jan. 2004. ISSN 1046-8188. Disponível em: <https://doi.org/10.1145/963770.963772>.

HIJIKATA, Y.; IWAHAMA, K. Content-based music filtering system with editable user profile. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2006. (SAC '06), p. 1050–1057. ISBN 1-59593-108-2. Disponível em: <http://doi.acm.org/10.1145/1141277.1141526>.

INNVONIX. *The Difference Between SaaS (Software as a Service) and SOA (Service-Oriented Architecture)*. 2019. Disponível em: <shorturl.at/fgu45>.

ISINKAYE, F.; FOLAJIMI, Y.; OJOKOH, B. Recommendation systems: Principles, methods and evaluation. *Egyptian Informatics Journal*, v. 16, n. 3, p. 261 – 273, 2015. ISSN 1110-8665. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1110866515000341>.

ISO/IEC/IEEE International Standard - Systems and software engineering â€" Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, p. 1–418, dez. 2010.

JAFARKARIMI, H.; SIM, A. T. H.; SAADATDOOST, R. A Naïve Recommendation Model for Large Databases. In: *International Journal of Information and Education Technology*. [S.l.: s.n.], 2012. vol. 2, p. 216 & 219.

JAIN, H.; KAKKAR, M. Job Recommendation System based on Machine Learning and Data Mining Techniques using RESTful API and Android IDE. In: *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. Noida, India: IEEE, 2019. p. 416–421. ISBN 9781538659335. Disponível em: <https://ieeexplore.ieee.org/document/8776964/>.

JENSON, G. *grahamjenson/list_of_recommender_systems*. 2020. Original-date: 2015-06-12T07:31:32Z. Disponível em: <https://github.com/grahamjenson/list_of_recommender_systems>.

KALE, V. Service-oriented architecture. In: _____. [S.l.: s.n.], 2018. p. 183–204. ISBN 9780429453311.

KAVITHA, M.; DAMODHARAN, P. Software as a service in cloud computing. *International Journal Of Recent Advances in Engineering Technology*, v. 08, p. 1–4, 04 2020.

KNAPP, B. *An introduction to IaaS (Infrastructure-as-a-Service), its components, advantages, pricing, and how it relates to PaaS, SaaS, BMaaS, containers, and serverless.* 2019. Disponível em: <https://www.ibm.com/cloud/learn/iaas>.

KOREN, Y.; BELL, R. Advances in collaborative filtering. In: _____. [S.l.: s.n.], 2015. p. 77–118. ISBN 978-1-4899-7636-9.

LOPS, P.; GEMMIS, M. de; SEMERARO, G. Content-based recommender systems: State of the art and trends. In: RICCI, F. et al. (Ed.). *Recommender Systems Handbook*. Springer US, 2011. p. 73–105. ISBN 978-0-387-85819-7. Disponível em: <http://dx.doi.org/10.1007/978-0-387-85820-3_3>.

LUCIAN, R. Repensando o uso da escala likert: Tradição ou escolha técnica? *PMKT - Revista Brasileira de Pesquisa de Marketing, Opinião e Mídia.*, v. 18, p. 13–32, 04 2016.

MACMANUS, R. *5 Problems of Recommender Systems*. 2009. Disponível em: <https://readwrite.com/5_problems_of_recommender_systems/>.

MARTIN-LOPEZ, A.; SEGURA, S.; RUIZ-CORTéS, A. Test coverage criteria for RESTful web APIs. In: *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. Tallinn Estonia: ACM, 2019. p. 15–21. ISBN 9781450368506. Disponível em: <https://dl.acm.org/doi/10.1145/3340433.3342822>.

MCNEE, S.; RIEDL, J.; KONSTAN, J. Being accurate is not enough: How accuracy metrics have hurt recommender systems. In: . [S.l.: s.n.], 2006. p. 1097–1101.

MELL, P. M.; GRANCE, T. *The NIST definition of cloud computing*. Gaithersburg, MD, 2011. NIST SP 800–145 p. Disponível em: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>.

MELVILLE, P.; SINDHWANI, V. Recommender Systems. In: *Encyclopedia of Machine Learning*. [S.l.: s.n.], 2010.

MITTELSTADT, B.; RUSSELL, C.; WACHTER, S. Explaining Explanations in AI. *Proceedings of the Conference on Fairness, Accountability, and Transparency*, p. 279–288, jan. 2019. ArXiv: 1811.01439. Disponível em: <http://arxiv.org/abs/1811.01439>.

MOONEY, R. J.; ROY, L. Content-based book recommendation using learning for text categorization. In: *Workshop Recom. Sys.: Algo. and Evaluation*. [S.l.: s.n.], 1999.

MOZILLA. *HTTP request methods - HTTP | MDN*. 2020. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>.

MOZILLA. *HTTP status codes - HTTP | MDN*. 2021. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>.

NADAREISHVILI, I. *Microservice architecture : Aligning principles, practices, and culture*. Sebastopol, CA: O'Reilly Media, 2016. ISBN 1491956259.

NEWMAN, S. *Building microservices: designing fine-grained systems*. First edition. Beijing Sebastopol, CA: O'Reilly Media, 2015. OCLC: ocn881657228. ISBN 9781491950357.

Okon E Uko; B O Eke; ASAGBA, P. O. An Improved Online Book Recommender System using Collaborative Filtering Algorithm. 2018. Disponível em: <http://rgdoi.net/10.13140/RG.2.2.24240.46086>.

PAUTASSO, C. et al. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software*, v. 34, n. 1, p. 91–98, jan. 2017. ISSN 0740-7459, 1937-4194. Disponível em: <https://ieeexplore.ieee.org/document/7819415/>.

PESKA, L.; VOJTAS, P. Enhancing Recommender System with Linked Open Data. In: *Proceedings of the 10th International Conference on Flexible Query Answering Systems - Volume 8132*. Berlin, Heidelberg: Springer-Verlag, 2013. (FQAS 2013), p. 483–494. ISBN 9783642407680. Disponível em: <https://doi.org/10.1007/978-3-642-40769-7_42>.

PHAN, M. *Coupling and cohesion in OOP*. 2019. Disponível em: <http://ducmanhphan.github.io/2019-03-23-Coupling-and-Cohension-in-OOP/>.

QIN, Y. *A Historical Survey of Music Recommendation Systems: Towards Evaluation*. Tese (Doutorado), 2013.

RAUF, I.; TROUBITSYNA, E.; PORRES, I. A systematic mapping study of API usability evaluation methods. *Computer Science Review*, v. 33, p. 49–68, ago. 2019. ISSN 15740137. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1574013718301515>.

RECOMMENDERSYSTEMS.COM. *Recommendations-As-a-Service (RaaS) – RS_c*. 2020. Disponível em: <https://recommender-systems.com/resources/recommendations-as-a-service-raas/>.

RESNICK, P.; VARIAN, H. R. Recommender systems. *Commun. ACM*, ACM, New York, NY, USA, v. 40, n. 3, p. 56–58, mar. 1997. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/245108.245121>.

RICCI, F.; ROKACH, L.; SHAPIRA, B. Introduction to Recommender Systems Handbook. In: RICCI, F. et al. (Ed.). *Recommender Systems Handbook*. Boston, MA: Springer US, 2011. p. 1–35. ISBN 9780387858203. Disponível em: <https://doi.org/10.1007/978-0-387-85820-3_1>.

RICCI, F.; ROKACH, L.; SHAPIRA, B. Introduction to recommender systems handbook. In: *Recommender systems handbook*. [S.l.]: Springer, 2011. p. 1–35.

RIGHTBRAINNETWORKS. *Why SaaS and Microservices are Critical to Developing in the Cloud – RightBrain Networks*. 2015. Disponível em: <https://www.rightbrainnetworks.com/2015/01/29/why-saas-and-microservices-are-critical-to-developing-in-the-cloud/>.

RUDMARK, D. The practices of unpaid third-party developers – implications for api design. In: . [S.l.: s.n.], 2013. v. 5.

RUNGTA, K. *RESTful Web Services Tutorial with REST API Example*. 2020. Disponível em: <https://www.guru99.com/restful-web-services.html>.

SCHAFER, B. et al. Collaborative filtering recommender systems. In: . [S.l.: s.n.], 2007.

SILVEIRA, T. et al. How good your recommender system is? a survey on evaluations in recommendation. *International Journal of Machine Learning and Cybernetics*, v. 10, 12 2017.

SINGH, S. K. et al. Software as a service. In: ______. [S.l.: s.n.], 2019. p. 95–118. ISBN 978-93-8817-666-8.

SONG, H. et al. Customizing multi-tenant saas by microservices: A reference architecture. In: *2019 IEEE International Conference on Web Services (ICWS)*. [S.l.: s.n.], 2019. p. 446–448.

STECK, H. Calibrated recommendations. In: *Proceedings of the 12th ACM Conference on Recommender Systems*. New York, NY, USA: Association for Computing Machinery, 2018. (RecSys '18), p. 154–162. ISBN 9781450359016. Disponível em: <https://doi.org/10.1145/3240323.3240372>.

SU, X.; KHOSHGOFTAAR, T. A survey of collaborative filtering techniques. *Adv. Artificial Intellegence*, v. 2009, 10 2009.

TECHCELLO. *Why Microservices adoption is crucial for SaaS companies*. 2020. Disponível em: <https://www.techcello.com/Why-Micro-Services-adoption-is-crucial-for-SaaS-companies/>.

TERUYA, H. S. et al. URecommender: An API for Recommendation Systems. In: *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*. Sevilla, Spain: IEEE, 2020. p. 1–6. ISBN 9789895465903. Disponível em: <https://ieeexplore.ieee.org/document/9141055/>.

VINEELA, A. et al. A Comprehensive Study and Evaluation of Recommender Systems. In: CHOWDARY, P. S. R. et al. (Ed.). *Microelectronics, Electromagnetics and Telecommunications*. Singapore: Springer, 2021. p. 45–53. ISBN 9789811538285.

WATTS, S.; HAZA, M. *SaaS vs PaaS vs IaaS: What's The Difference & How To Choose*. 2019. Disponível em: <https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/>.

WOLFF, E. *Microservices: flexible software architecture*. Boston: Addison-Wesley, 2017. OCLC: ocn965730846. ISBN 9780134602417.

ZANARDI, V.; CAPRA, L. Dynamic updating of online recommender systems via feed-forward controllers. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. New York, NY, USA: Association for Computing Machinery, 2011. (SEAMS '11), p. 11–19. ISBN 9781450305754. Disponível em: <https://doi.org/10.1145/1988008.1988011>.

ZHANG, Z. *zhenghaoz/gorse*. 2021. Original-date: 2018-08-14T11:01:09Z. Disponível em: <https://github.com/zhenghaoz/gorse>.

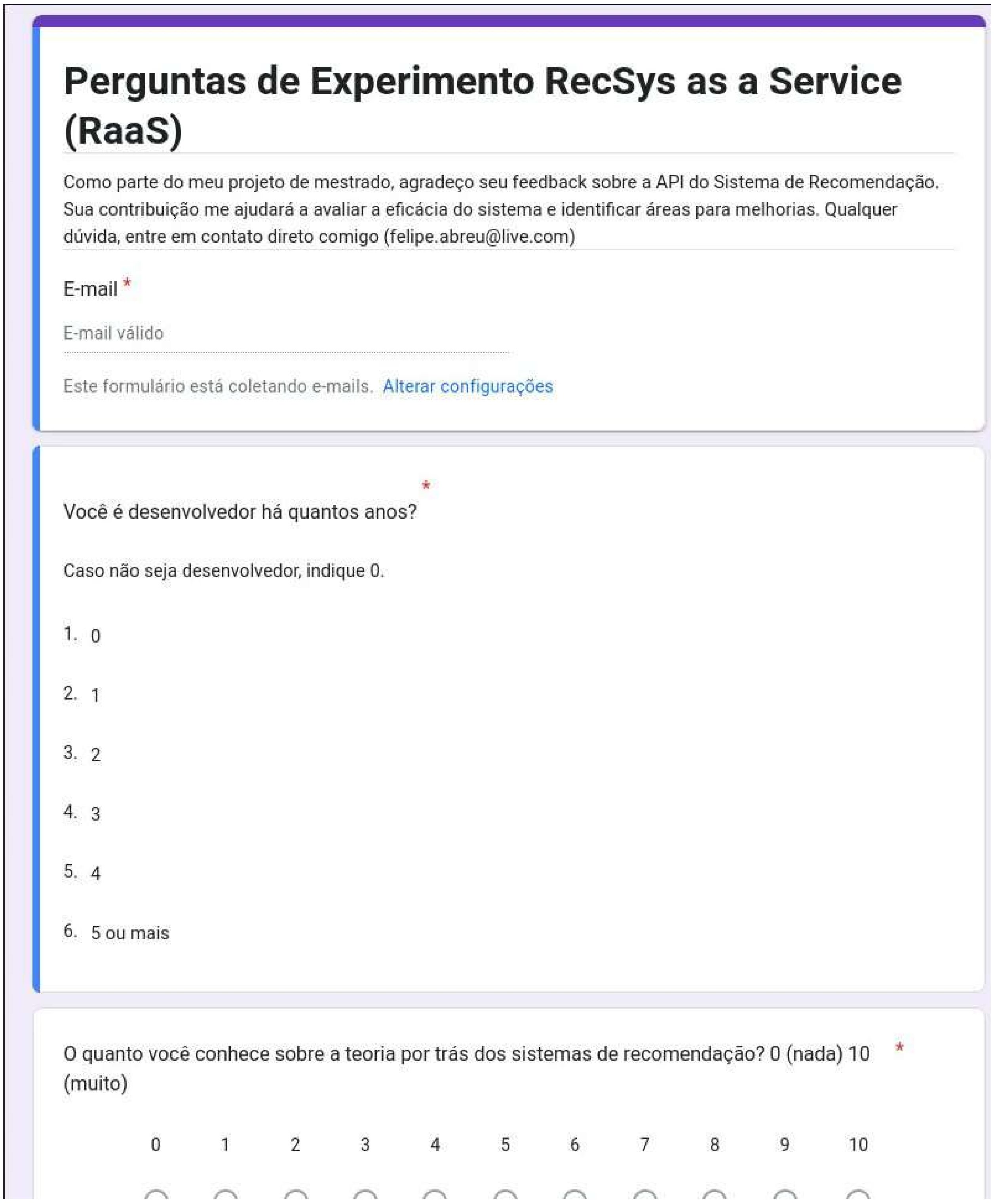
ZIBRAN, M. F.; EISHITA, F. Z.; ROY, C. K. Useful, But Usable? Factors Affecting the Usability of APIs. In: *2011 18th Working Conference on Reverse Engineering*. Limerick: IEEE, 2011. p. 151–155. ISBN 9781457719486. Disponível em: <http://ieeexplore.ieee.org/document/6079520/>.

ÇANO, E. *Cloud-based Recommendation Systems: Applications and Solution*. 2017. Disponível em: <https://iris.polito.it/retrieve/handle/11583/2669861/149344/ErionCanoCRSAS.pdf>.

Appendix
A

APPENDIX

A.1 EXPERIMENT FORM: HTTPS://FORMS.GLE/ODGHEYTJKH3BYXQU8



Perguntas de Experimento RecSys as a Service (RaaS)

Como parte do meu projeto de mestrado, agradeço seu feedback sobre a API do Sistema de Recomendação. Sua contribuição me ajudará a avaliar a eficácia do sistema e identificar áreas para melhorias. Qualquer dúvida, entre em contato direto comigo (felipe.abreu@live.com)

E-mail *

E-mail válido

Este formulário está coletando e-mails. Alterar configurações

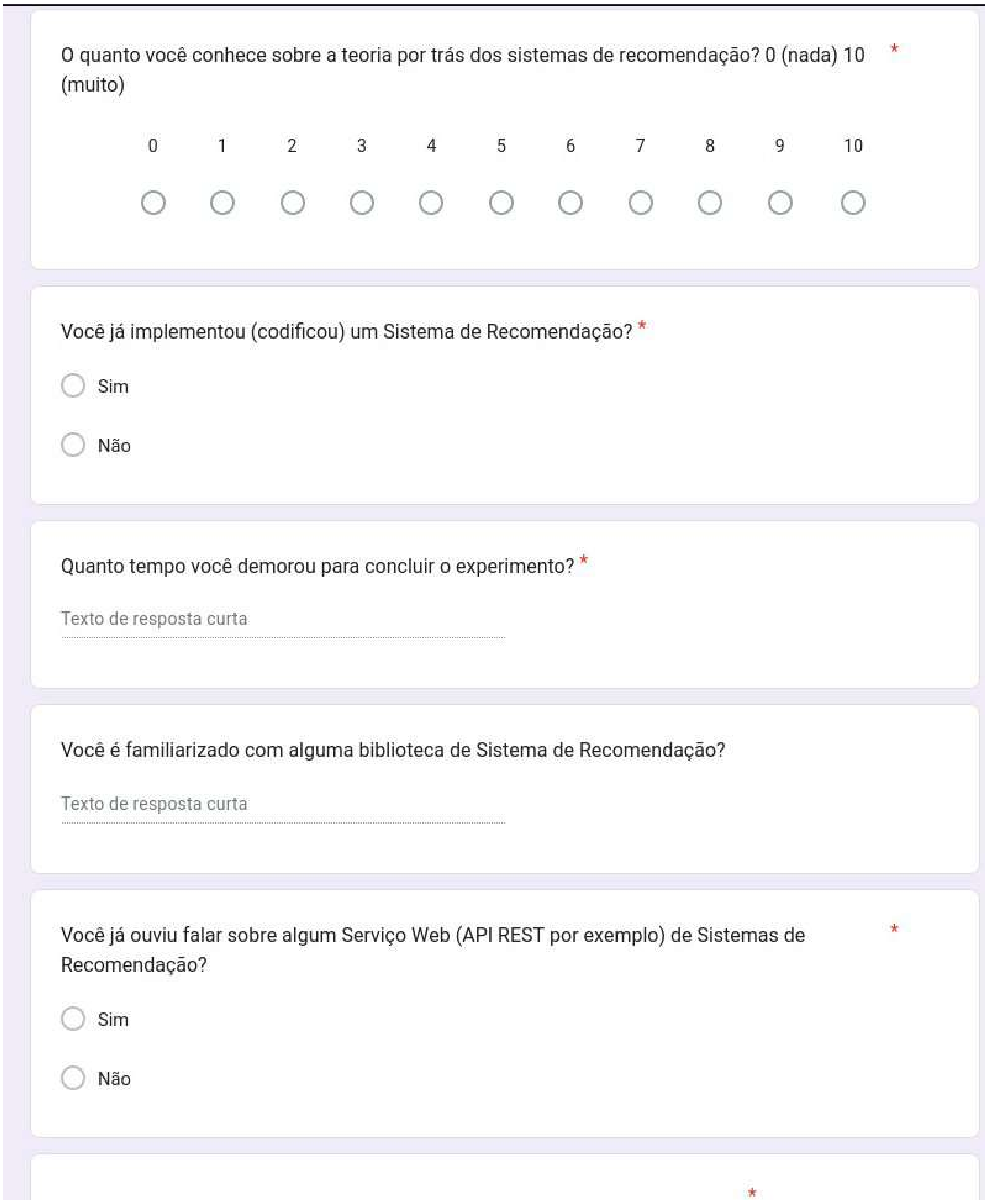
Você é desenvolvedor há quantos anos? *

Caso não seja desenvolvedor, indique 0.

1. 0
2. 1
3. 2
4. 3
5. 4
6. 5 ou mais

O quanto você conhece sobre a teoria por trás dos sistemas de recomendação? 0 (nada) 10 (muito) *

0	1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○	○



O quanto você conhece sobre a teoria por trás dos sistemas de recomendação? 0 (nada) 10 (muito) *

0	1	2	3	4	5	6	7	8	9	10
○	○	○	○	○	○	○	○	○	○	○

Você já implementou (codificou) um Sistema de Recomendação? *

○ Sim

○ Não

Quanto tempo você demorou para concluir o experimento? *

Texto de resposta curta

Você é familiarizado com alguma biblioteca de Sistema de Recomendação?

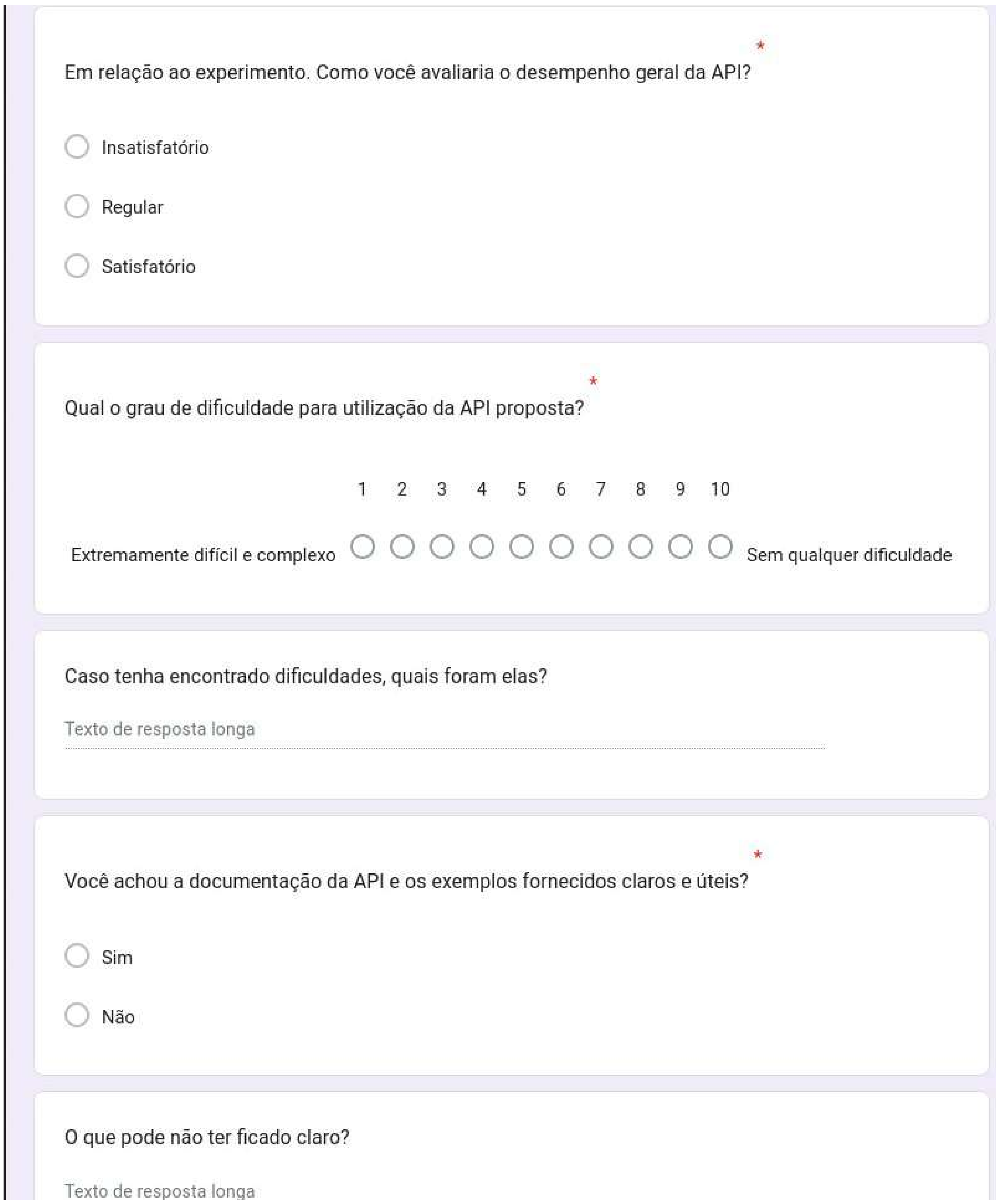
Texto de resposta curta

Você já ouviu falar sobre algum Serviço Web (API REST por exemplo) de Sistemas de Recomendação? *

○ Sim

○ Não

*



Em relação ao experimento. Como você avaliaria o desempenho geral da API? *

○ Insatisfatório

○ Regular

○ Satisfatório

Qual o grau de dificuldade para utilização da API proposta? *

	1	2	3	4	5	6	7	8	9	10	
Extremamente difícil e complexo	○	○	○	○	○	○	○	○	○	○	Sem qualquer dificuldade

Caso tenha encontrado dificuldades, quais foram elas?

Texto de resposta longa

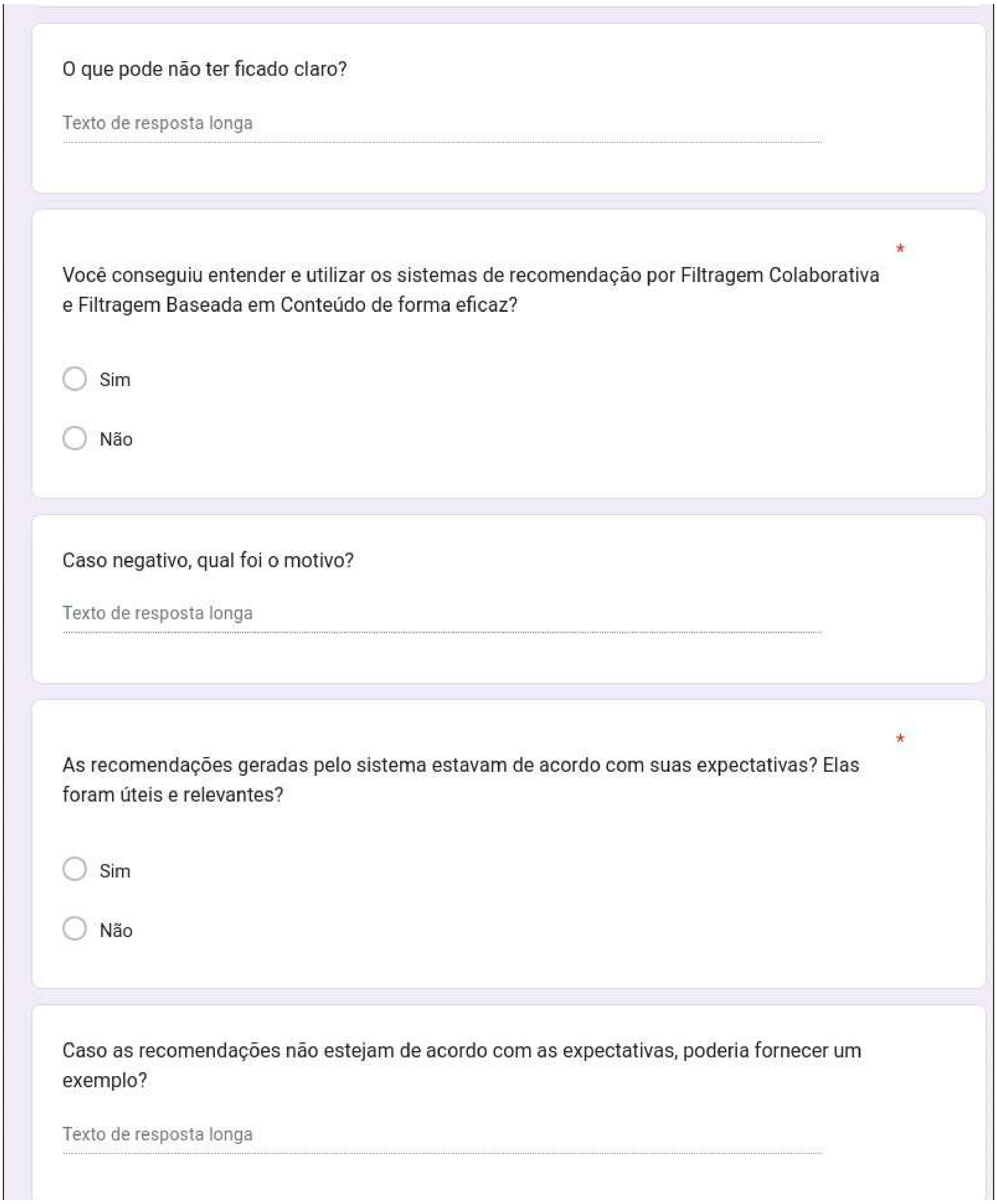
Você achou a documentação da API e os exemplos fornecidos claros e úteis? *

○ Sim

○ Não

O que pode não ter ficado claro?

Texto de resposta longa


O que pode não ter ficado claro?

Texto de resposta longa

Você conseguiu entender e utilizar os sistemas de recomendação por Filtragem Colaborativa e Filtragem Baseada em Conteúdo de forma eficaz? *

○ Sim

○ Não

Caso negativo, qual foi o motivo?

Texto de resposta longa

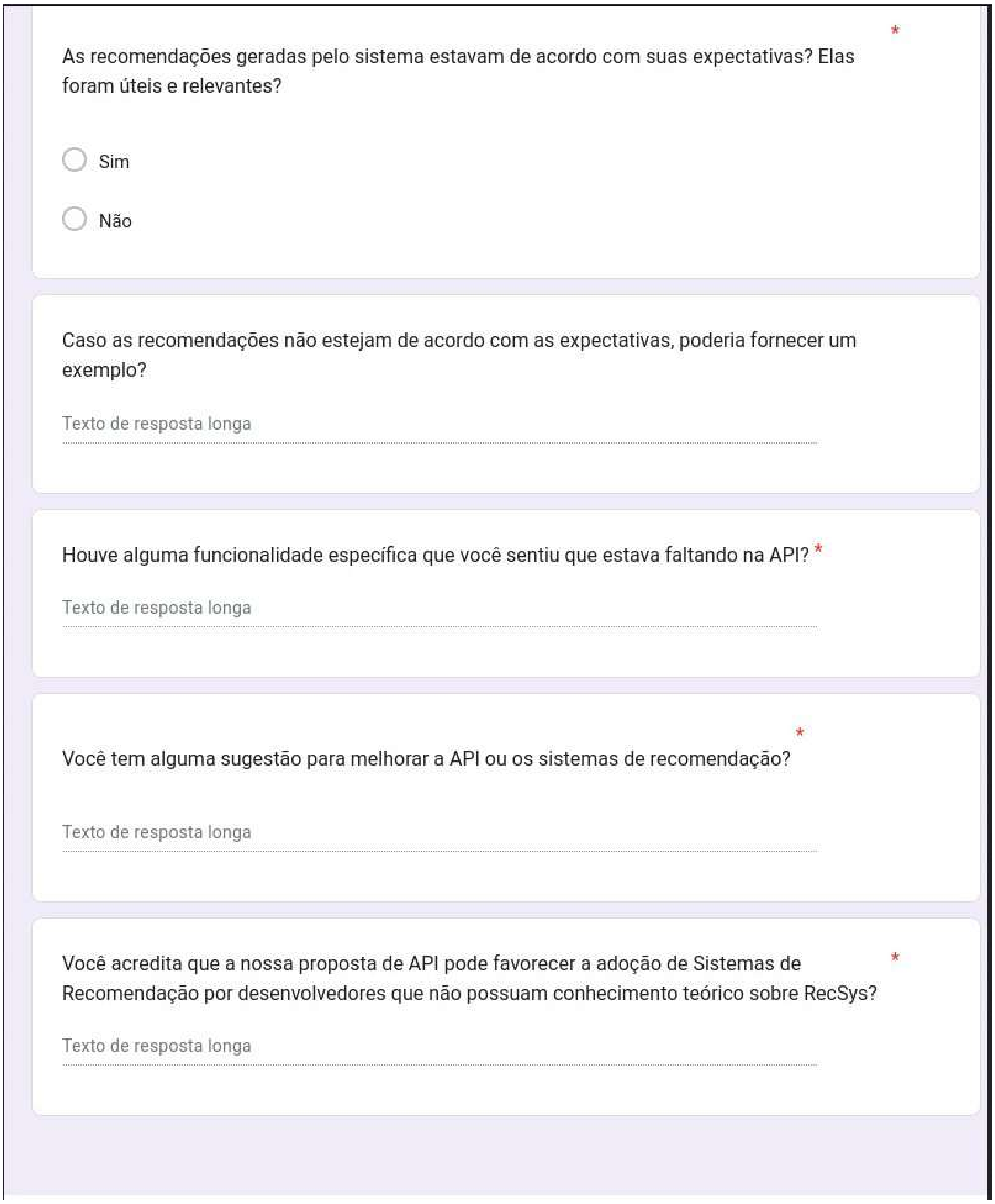
As recomendações geradas pelo sistema estavam de acordo com suas expectativas? Elas foram úteis e relevantes? *

○ Sim

○ Não

Caso as recomendações não estejam de acordo com as expectativas, poderia fornecer um exemplo?

Texto de resposta longa



As recomendações geradas pelo sistema estavam de acordo com suas expectativas? Elas foram úteis e relevantes? *

○ Sim

○ Não

Caso as recomendações não estejam de acordo com as expectativas, poderia fornecer um exemplo?

Texto de resposta longa

Houve alguma funcionalidade específica que você sentiu que estava faltando na API? *

Texto de resposta longa

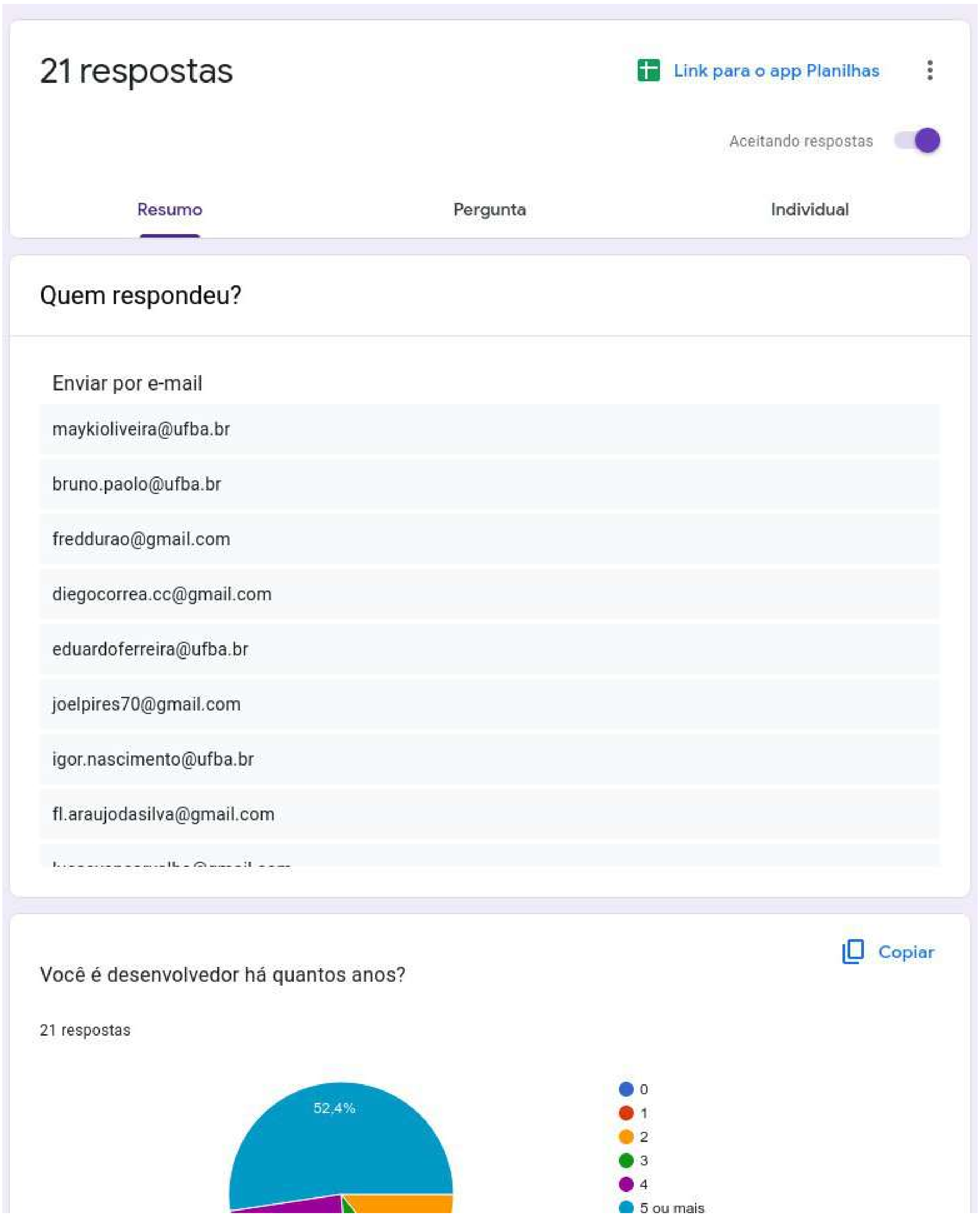
Você tem alguma sugestão para melhorar a API ou os sistemas de recomendação? *

Texto de resposta longa

Você acredita que a nossa proposta de API pode favorecer a adoção de Sistemas de Recomendação por desenvolvedores que não possuam conhecimento teórico sobre RecSys? *

Texto de resposta longa

A.2 EXPERIMENT RESPONSES: HTTPS://FORMS.GLE/ODGHEYTJKH3BYXQU8



21 respostas

Link para o app Planilhas

Aceitando respostas

Resumo Pergunta Individual

Quem respondeu?

Enviar por e-mail

maykioliveira@ufba.br

bruno.paolo@ufba.br

freddurao@gmail.com

diegocorrea.cc@gmail.com

eduardoferreira@ufba.br

joelpires70@gmail.com

igor.nascimento@ufba.br

fl.araujodasilva@gmail.com

Copiar

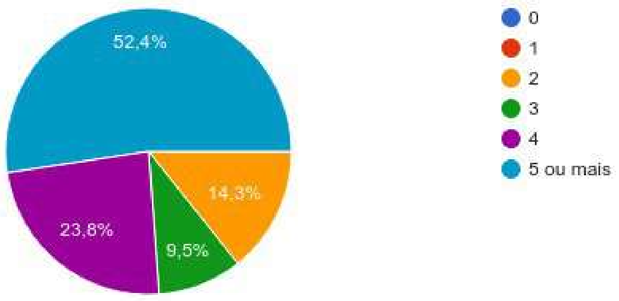
Você é desenvolvedor há quantos anos?

21 respostas

Copiar

Você é desenvolvedor há quantos anos?

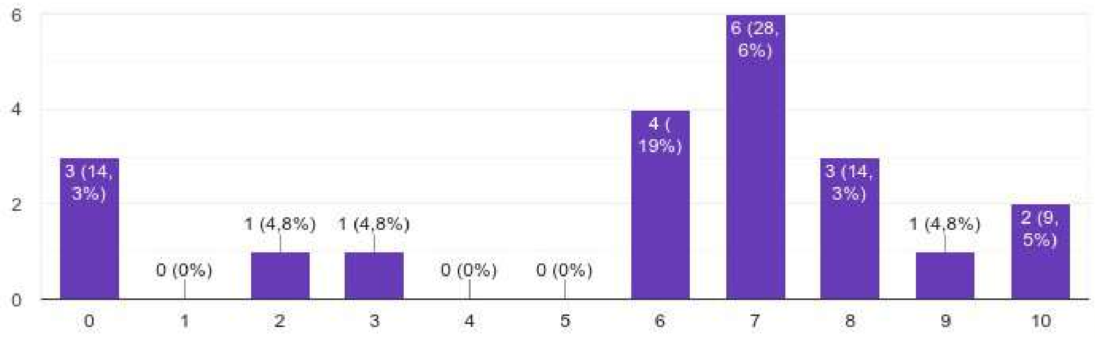
21 respostas



Copiar

O quanto você conhece sobre a teoria por trás dos sistemas de recomendação? 0 (nada) 10 (muito)

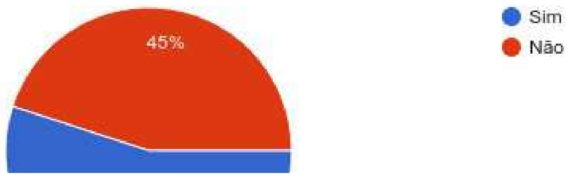
21 respostas



Copiar

Você já implementou (codificou) um Sistema de Recomendação?

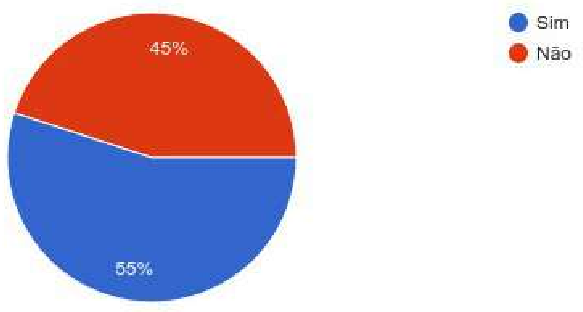
20 respostas



Você já implementou (codificou) um Sistema de Recomendação?

Copiar

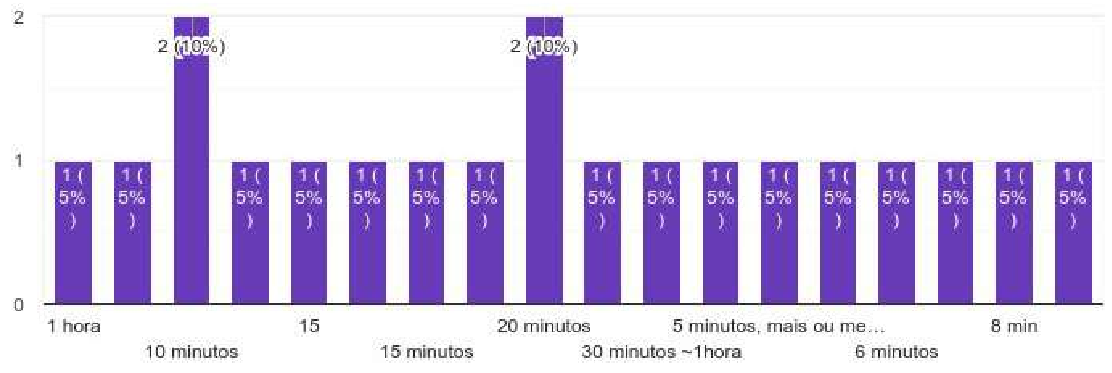
20 respostas



Quanto tempo você demorou para concluir o experimento?

Copiar

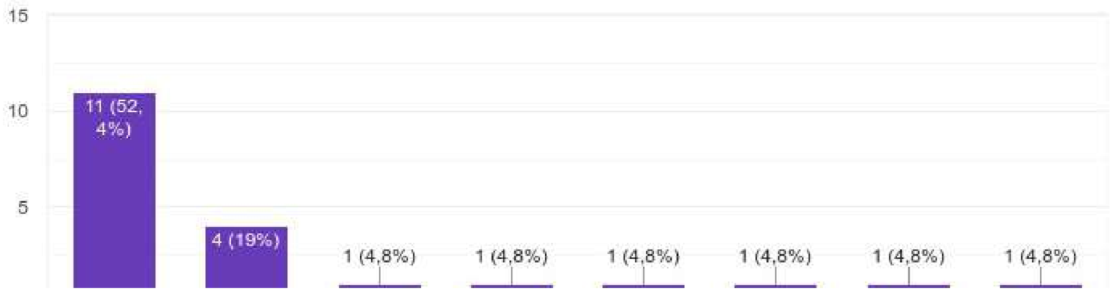
20 respostas



Você é familiarizado com alguma biblioteca de Sistema de Recomendação?

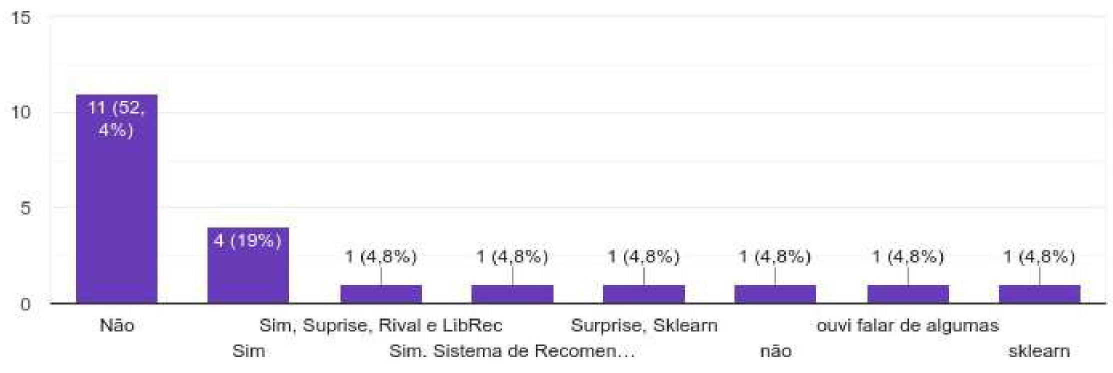
Copiar

21 respostas



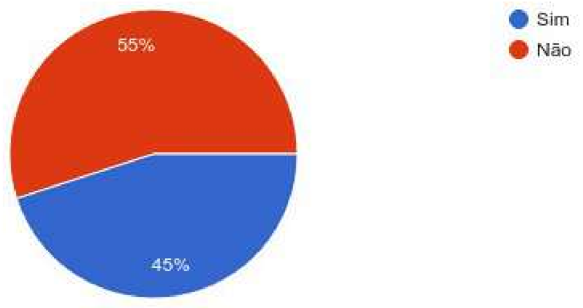
Você é familiarizado com alguma biblioteca de Sistema de Recomendação?

21 respostas



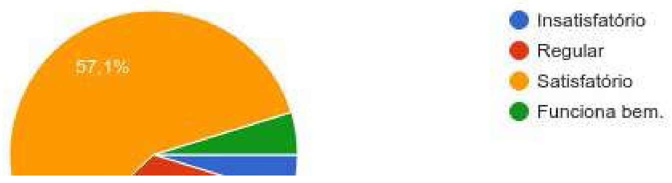
Você já ouviu falar sobre algum Serviço Web (API REST por exemplo) de Sistemas de Recomendação?

20 respostas



Em relação ao experimento. Como você avaliaria o desempenho geral da API?

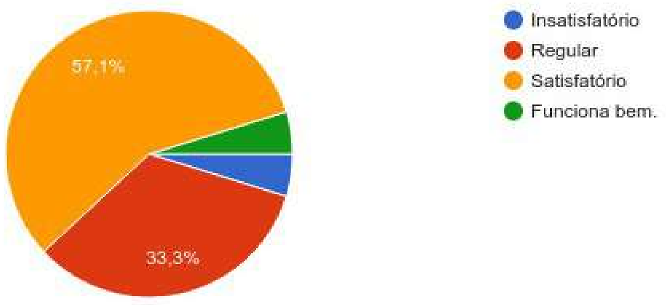
21 respostas



Em relação ao experimento. Como você avaliaria o desempenho geral da API?

Copiar

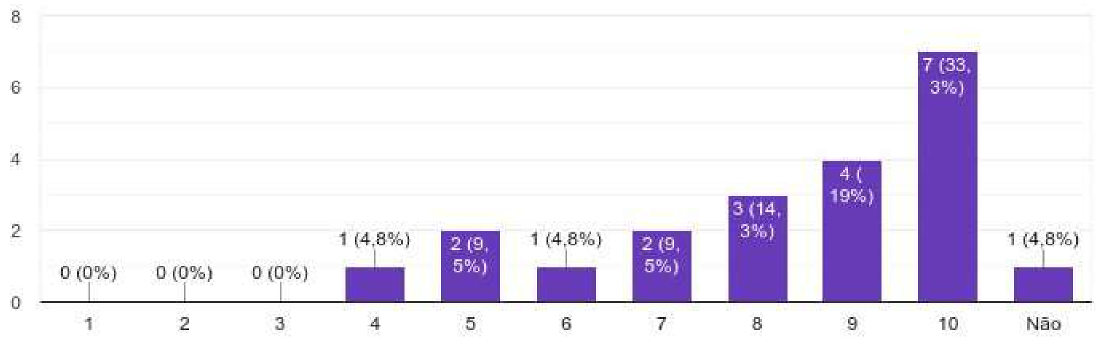
21 respostas



Qual o grau de dificuldade para utilização da API proposta?

Copiar

21 respostas



Caso tenha encontrado dificuldades, quais foram elas?

12 respostas

O endpoint de criação de usuários não aceita body

Não foram encontradas dificuldades

Não encontrei dificuldades, entretanto, o sistema falhou no passo PASSO 7.1. Recebo "Internal Server Error"

Caso tenha encontrado dificuldades, quais foram elas?

12 respostas

O endpoint de criação de usuários não aceita body

Não foram encontradas dificuldades

Não encontrei dificuldades, entretanto, o sistema falhou no passo PASSO 7.1. Recebo "Internal Server Error" como resposta.

- Alguns endpoints no Swagger estavam incompletos, não podendo passar parâmetros, o que dificultou a execução
- A execução do endpoint GET /user/recommendations retornou um erro 500, mesmo seguindo todos os passos do experimento

O comando do passo 7.1 não funcionou para mim

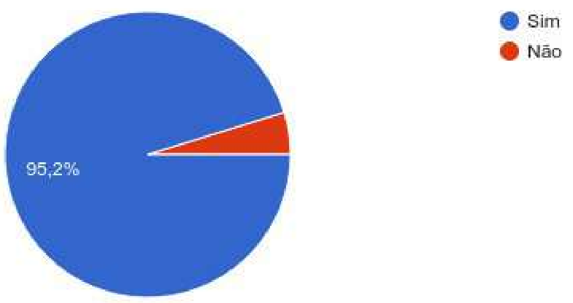
Minha única dificuldade foi entender como funciona a filtragem colaborativa. Tentei alguns dados nos parâmetros e apenas deu status 500, não consegui utilizar o endpoint "user/recommendations".

A documentação tem algumas coisas imprecisas.

Copiar

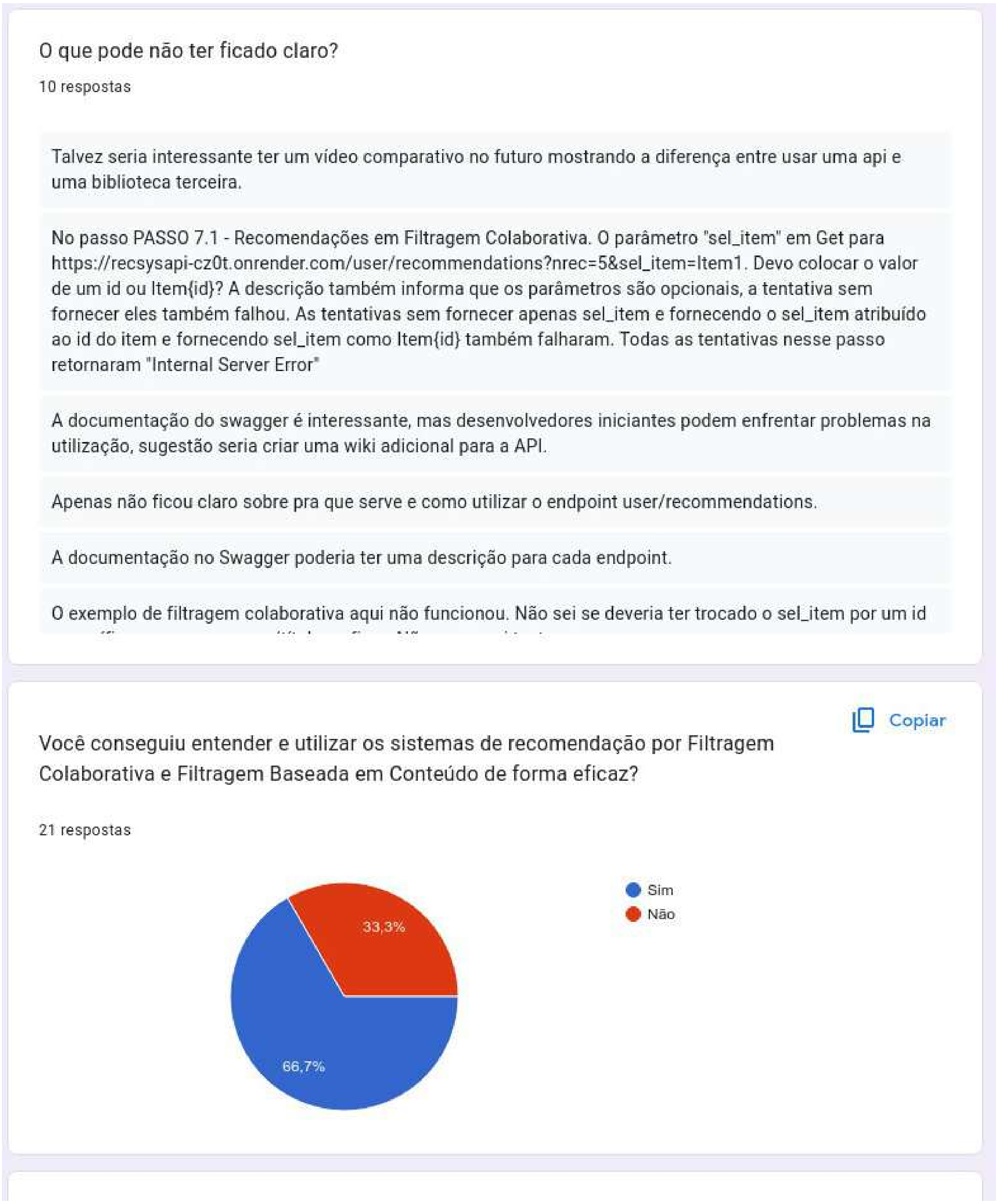
Você achou a documentação da API e os exemplos fornecidos claros e úteis?

21 respostas



O que pode não ter ficado claro?

10 respostas



O que pode não ter ficado claro?

10 respostas

Talvez seria interessante ter um vídeo comparativo no futuro mostrando a diferença entre usar uma api e uma biblioteca terceira.

No passo PASSO 7.1 - Recomendações em Filtragem Colaborativa. O parâmetro "sel_item" em Get para https://recsysapi-cz0t.onrender.com/user/recommendations?nrec=5&sel_item=Item1. Devo colocar o valor de um id ou Item{id}? A descrição também informa que os parâmetros são opcionais, a tentativa sem fornecer eles também falhou. As tentativas sem fornecer apenas sel_item e fornecendo o sel_item atribuído ao id do item e fornecendo sel_item como Item{id} também falharam. Todas as tentativas nesse passo retornaram "Internal Server Error"

A documentação do swagger é interessante, mas desenvolvedores iniciantes podem enfrentar problemas na utilização, sugestão seria criar uma wiki adicional para a API.

Apenas não ficou claro sobre pra que serve e como utilizar o endpoint user/recommendations.

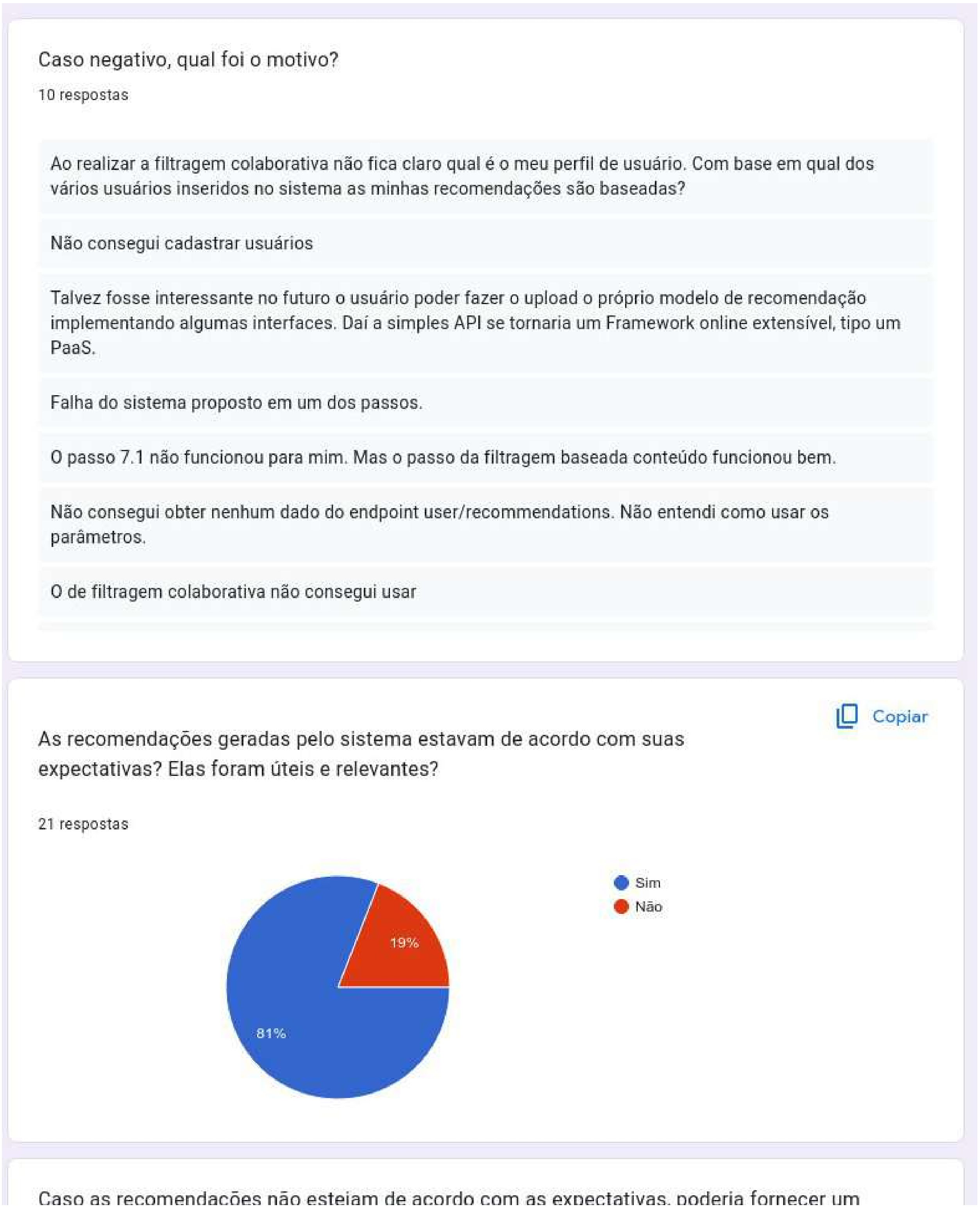
A documentação no Swagger poderia ter uma descrição para cada endpoint.

O exemplo de filtragem colaborativa aqui não funcionou. Não sei se deveria ter trocado o sel_item por um id

Copiar

Você conseguiu entender e utilizar os sistemas de recomendação por Filtragem Colaborativa e Filtragem Baseada em Conteúdo de forma eficaz?

21 respostas



Caso negativo, qual foi o motivo?

10 respostas

Ao realizar a filtragem colaborativa não fica claro qual é o meu perfil de usuário. Com base em qual dos vários usuários inseridos no sistema as minhas recomendações são baseadas?

Não consegui cadastrar usuários

Talvez fosse interessante no futuro o usuário poder fazer o upload o próprio modelo de recomendação implementando algumas interfaces. Daí a simples API se tornaria um Framework online extensível, tipo um PaaS.

Falha do sistema proposto em um dos passos.

O passo 7.1 não funcionou para mim. Mas o passo da filtragem baseada conteúdo funcionou bem.

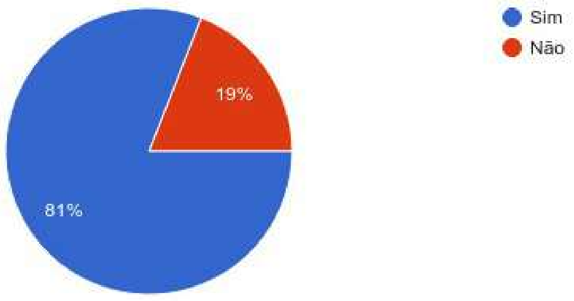
Não consegui obter nenhum dado do endpoint user/recommendations. Não entendi como usar os parâmetros.

O de filtragem colaborativa não consegui usar

Copiar

As recomendações geradas pelo sistema estavam de acordo com suas expectativas? Elas foram úteis e relevantes?

21 respostas

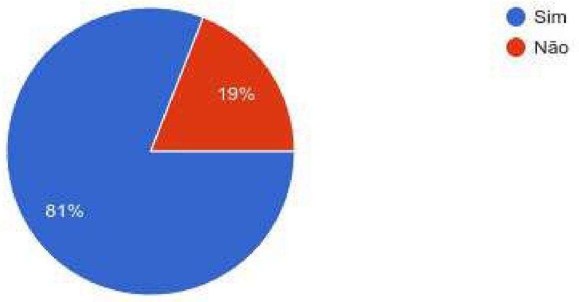


Caso as recomendações não estejam de acordo com as expectativas, poderia fornecer um

Copiar

As recomendações geradas pelo sistema estavam de acordo com suas expectativas? Elas foram úteis e relevantes?

21 respostas



Caso as recomendações não estejam de acordo com as expectativas, poderia fornecer um exemplo?

2 respostas

Recomendações foram precisas e satisfatórias.

item/neighbors?itemno=2&nitems=5, na segunda posição aparece a Torre de Pisa e na terceira o Palácio de Versalhes. Não entendi se o palácio não deveria estar mais relacionado à Torre Eiffel por estarem na França ou se a Torre de Pisa está mais próxima por ser também uma torre.

Houve alguma funcionalidade específica que você sentiu que estava faltando na API?

21 respostas

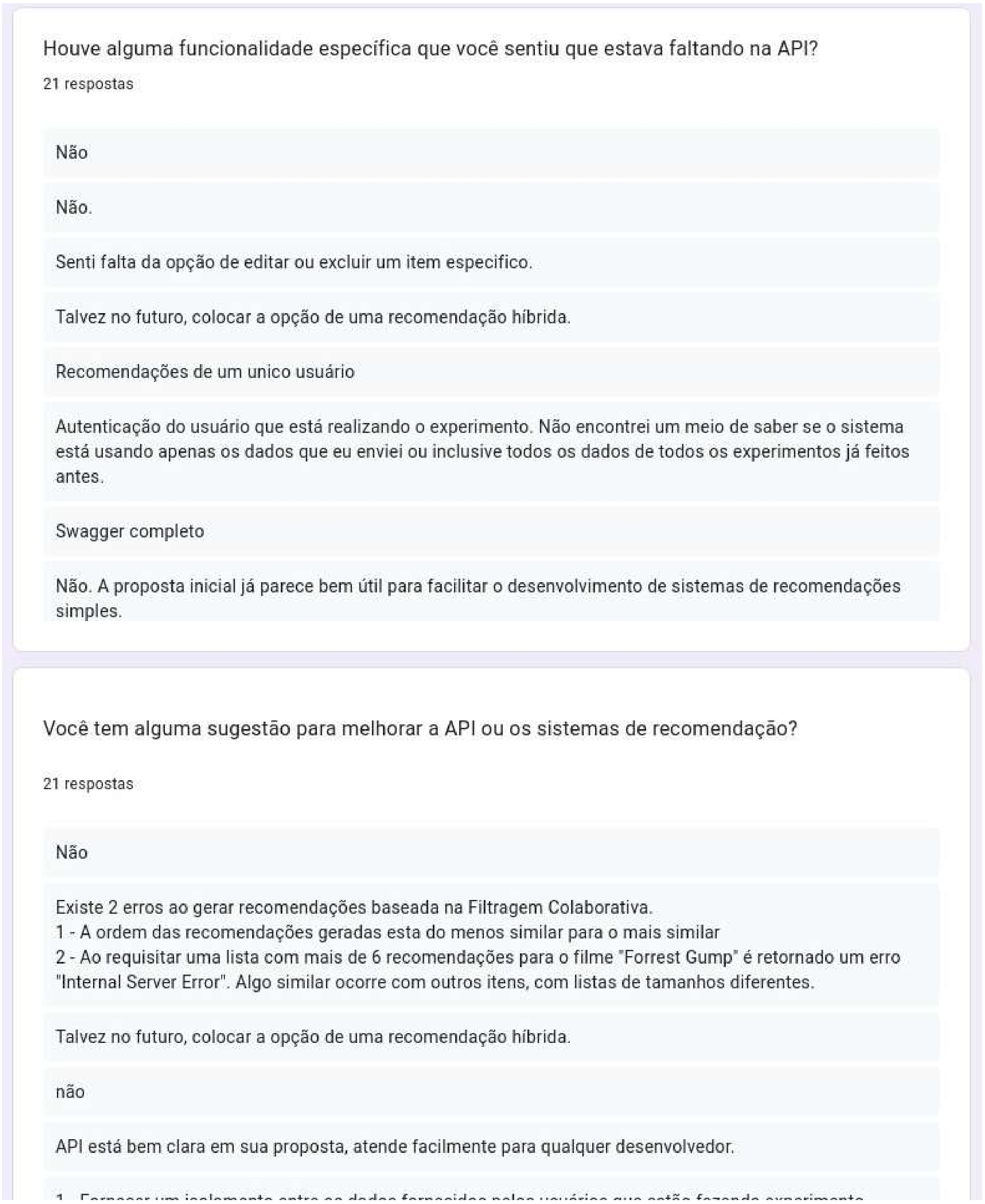
Não

Não.

Senti falta da opção de editar ou excluir um item especifico.

Talvez no futuro, colocar a opção de uma recomendação híbrida.

Recomendacões de um unico usuário



Houve alguma funcionalidade específica que você sentiu que estava faltando na API?

21 respostas

Não

Não.

Senti falta da opção de editar ou excluir um item especifico.

Talvez no futuro, colocar a opção de uma recomendação híbrida.

Recomendações de um unico usuário

Autenticação do usuário que está realizando o experimento. Não encontrei um meio de saber se o sistema está usando apenas os dados que eu enviei ou inclusive todos os dados de todos os experimentos já feitos antes.

Swagger completo

Não. A proposta inicial já parece bem útil para facilitar o desenvolvimento de sistemas de recomendações simples.

Você tem alguma sugestão para melhorar a API ou os sistemas de recomendação?

21 respostas

Não

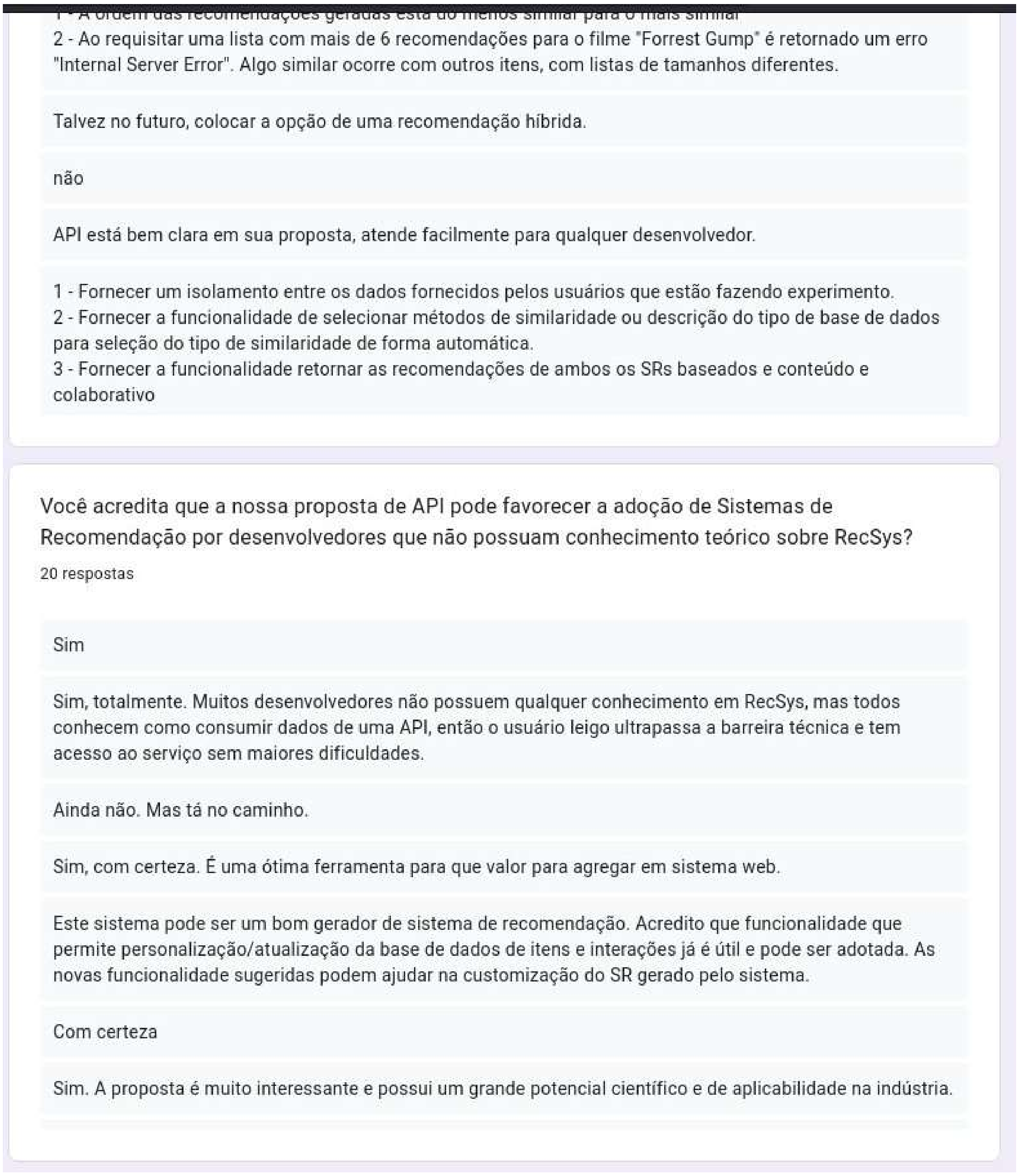
Existe 2 erros ao gerar recomendações baseada na Filtragem Colaborativa.
1 - A ordem das recomendações geradas esta do menos similar para o mais similar
2 - Ao requisitar uma lista com mais de 6 recomendações para o filme "Forrest Gump" é retornado um erro "Internal Server Error". Algo similar ocorre com outros itens, com listas de tamanhos diferentes.

Talvez no futuro, colocar a opção de uma recomendação híbrida.

não

API está bem clara em sua proposta, atende facilmente para qualquer desenvolvedor.

1 - Fornecer um isolamento entre os dados fornecidos pelos usuários que estão fazendo experimento



1 - A ordem das recomendações geradas está do menos similar para o mais similar
2 - Ao requisitar uma lista com mais de 6 recomendações para o filme "Forrest Gump" é retornado um erro "Internal Server Error". Algo similar ocorre com outros itens, com listas de tamanhos diferentes.

Talvez no futuro, colocar a opção de uma recomendação híbrida.

não

API está bem clara em sua proposta, atende facilmente para qualquer desenvolvedor.

1 - Fornecer um isolamento entre os dados fornecidos pelos usuários que estão fazendo experimento.
2 - Fornecer a funcionalidade de selecionar métodos de similaridade ou descrição do tipo de base de dados para seleção do tipo de similaridade de forma automática.
3 - Fornecer a funcionalidade retornar as recomendações de ambos os SRs baseados e conteúdo e colaborativo

Você acredita que a nossa proposta de API pode favorecer a adoção de Sistemas de Recomendação por desenvolvedores que não possuam conhecimento teórico sobre RecSys?

20 respostas

Sim

Sim, totalmente. Muitos desenvolvedores não possuem qualquer conhecimento em RecSys, mas todos conhecem como consumir dados de uma API, então o usuário leigo ultrapassa a barreira técnica e tem acesso ao serviço sem maiores dificuldades.

Ainda não. Mas tá no caminho.

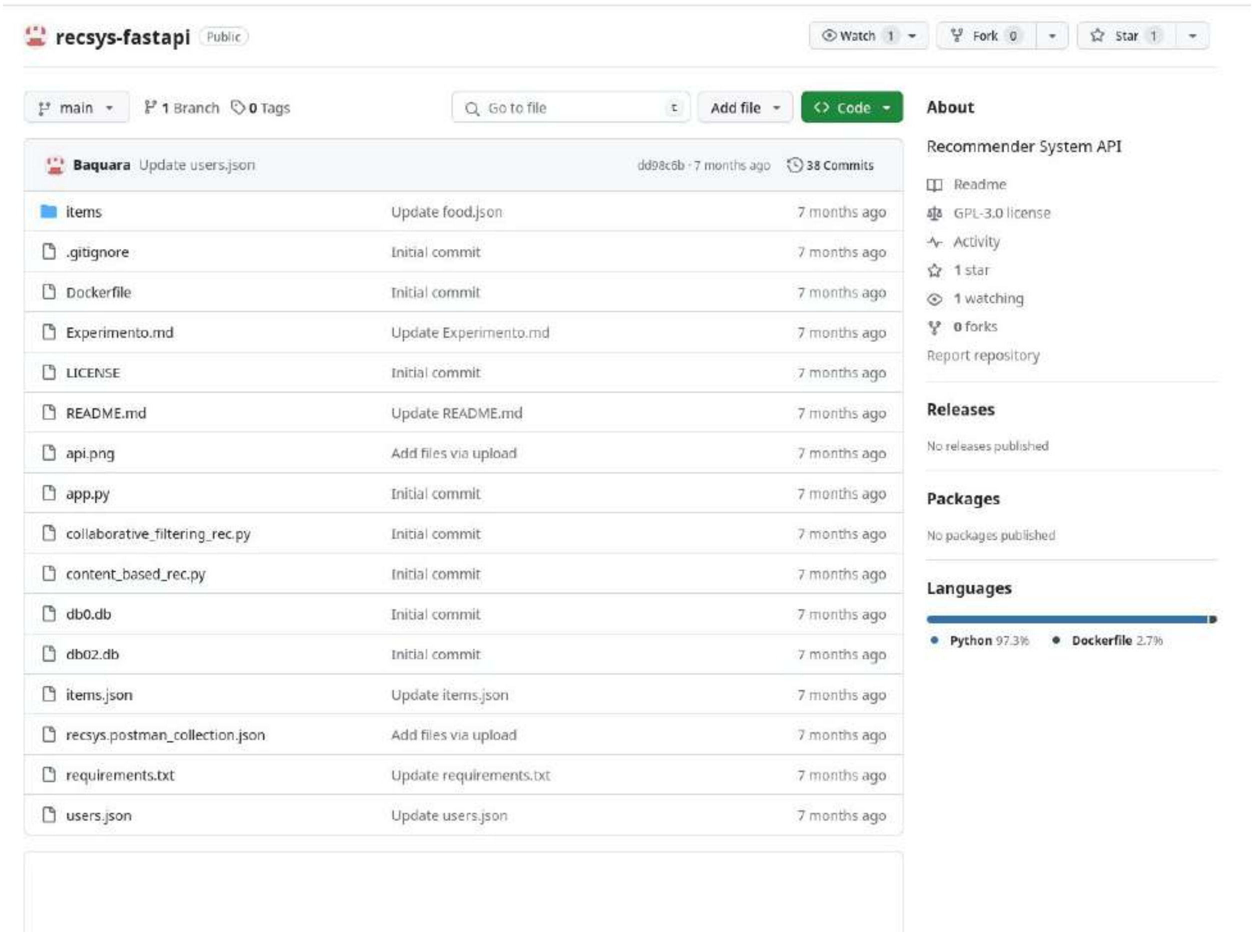
Sim, com certeza. É uma ótima ferramenta para que valor para agregar em sistema web.

Este sistema pode ser um bom gerador de sistema de recomendação. Acredito que funcionalidade que permite personalização/atualização da base de dados de itens e interações já é útil e pode ser adotada. As novas funcionalidade sugeridas podem ajudar na customização do SR gerado pelo sistema.

Com certeza

Sim. A proposta é muito interessante e possui um grande potencial científico e de aplicabilidade na indústria.

A.3 CODE REPOSITORY (GITHUB): HTTPS://GITHUB.COM/BAQUARA/RECSYS-FASTAPI/


recsys-fastapi Public

Watch 1 · Fork 0 · Star 1

main · 1 Branch · 0 Tags · Go to file · Add file · Code

Baquara Update users.json · dd98c6b · 7 months ago · 38 Commits

items	Update food.json	7 months ago
.gitignore	Initial commit	7 months ago
Dockerfile	Initial commit	7 months ago
Experimento.md	Update Experimento.md	7 months ago
LICENSE	Initial commit	7 months ago
README.md	Update README.md	7 months ago
api.png	Add files via upload	7 months ago
app.py	Initial commit	7 months ago
collaborative_filtering_rec.py	Initial commit	7 months ago
content_based_rec.py	Initial commit	7 months ago
db0.db	Initial commit	7 months ago
db02.db	Initial commit	7 months ago
items.json	Update items.json	7 months ago
recsys.postman_collection.json	Add files via upload	7 months ago
requirements.txt	Update requirements.txt	7 months ago
users.json	Update users.json	7 months ago

About

Recommender System API

Readme

GPL-3.0 license

Activity

1 star

1 watching

0 forks

Report repository

Releases

No releases published

Packages

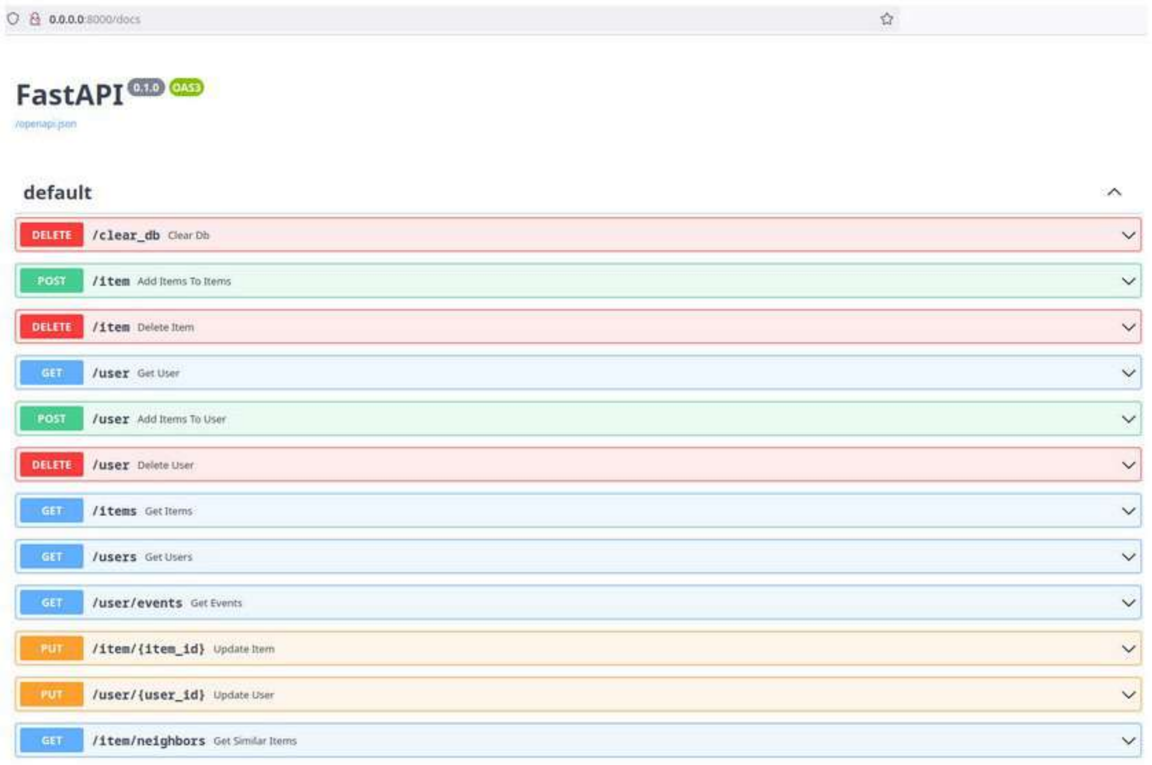
No packages published

Languages

● Python 97.3% ● Dockerfile 2.7%

Recommender System API

The Recommender System API is a general-purpose recommendation engine that provides API endpoints for managing items, users, events, and generating recommendations. It's built using Python's FastAPI and SQLite, and incorporates collaborative filtering and content-based recommendation models to provide real-time and personalized item recommendations.



0.0.0.0:8000/docs

FastAPI 0.1.0 OAS3

/openapi.json

default

DELETE /clear_db Clear Db

POST /item Add Items To Items

DELETE /item Delete Item

GET /user Get User

POST /user Add Items To User

DELETE /user Delete User

GET /items Get Items

GET /users Get Users

GET /user/events Get Events

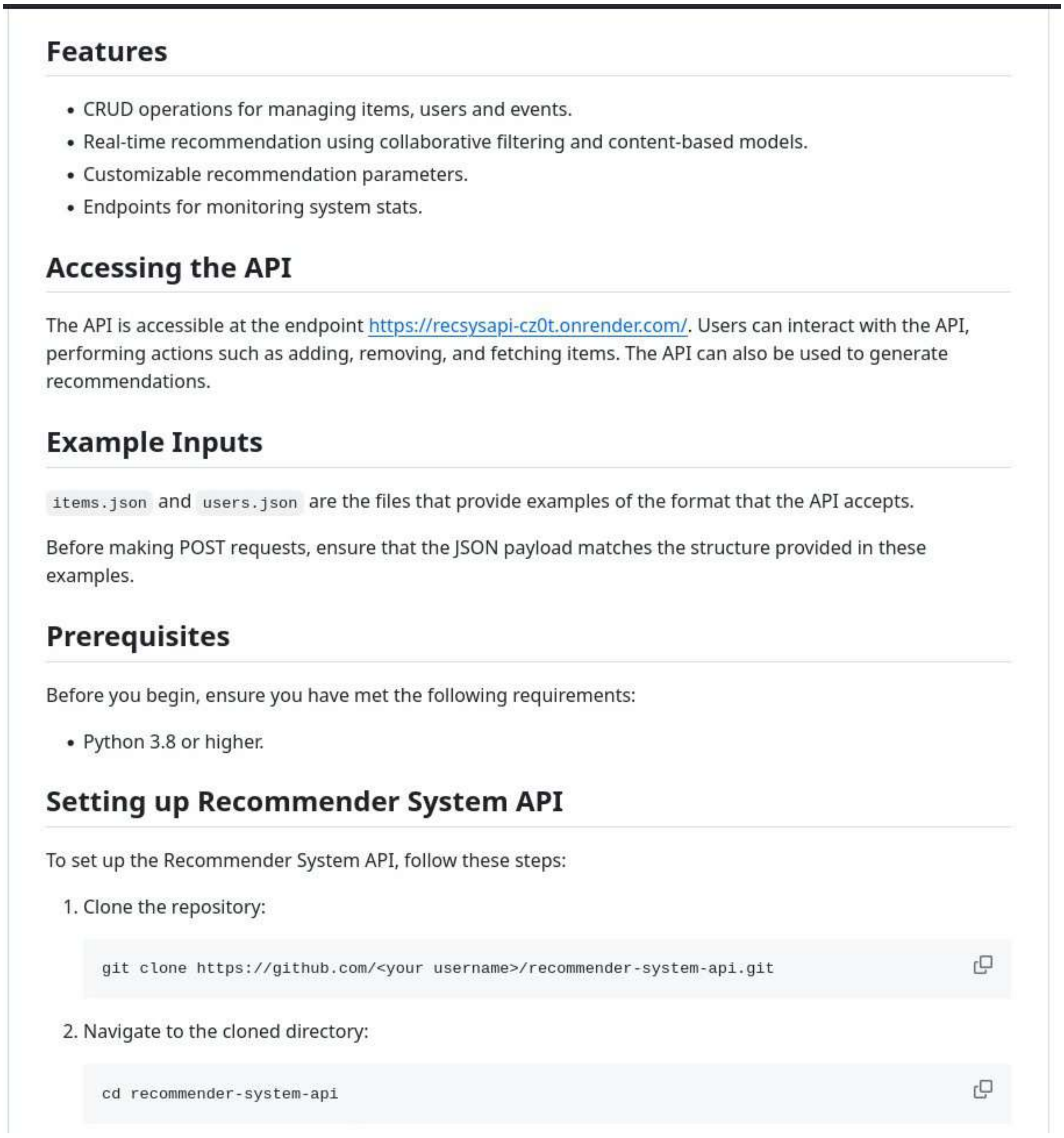
PUT /item/{item_id} Update Item

PUT /user/{user_id} Update User

GET /item/neighbors Get Similar Items

Features

- CRUD operations for managing items, users and events.
- Real-time recommendation using collaborative filtering and content-based models.
- Customizable recommendation parameters.
- Endpoints for monitoring system stats.



Features

- CRUD operations for managing items, users and events.
- Real-time recommendation using collaborative filtering and content-based models.
- Customizable recommendation parameters.
- Endpoints for monitoring system stats.

Accessing the API

The API is accessible at the endpoint https://recsysapi-cz0t.onrender.com/. Users can interact with the API, performing actions such as adding, removing, and fetching items. The API can also be used to generate recommendations.

Example Inputs

`items.json` and `users.json` are the files that provide examples of the format that the API accepts.

Before making POST requests, ensure that the JSON payload matches the structure provided in these examples.

Prerequisites

Before you begin, ensure you have met the following requirements:

- Python 3.8 or higher.

Setting up Recommender System API

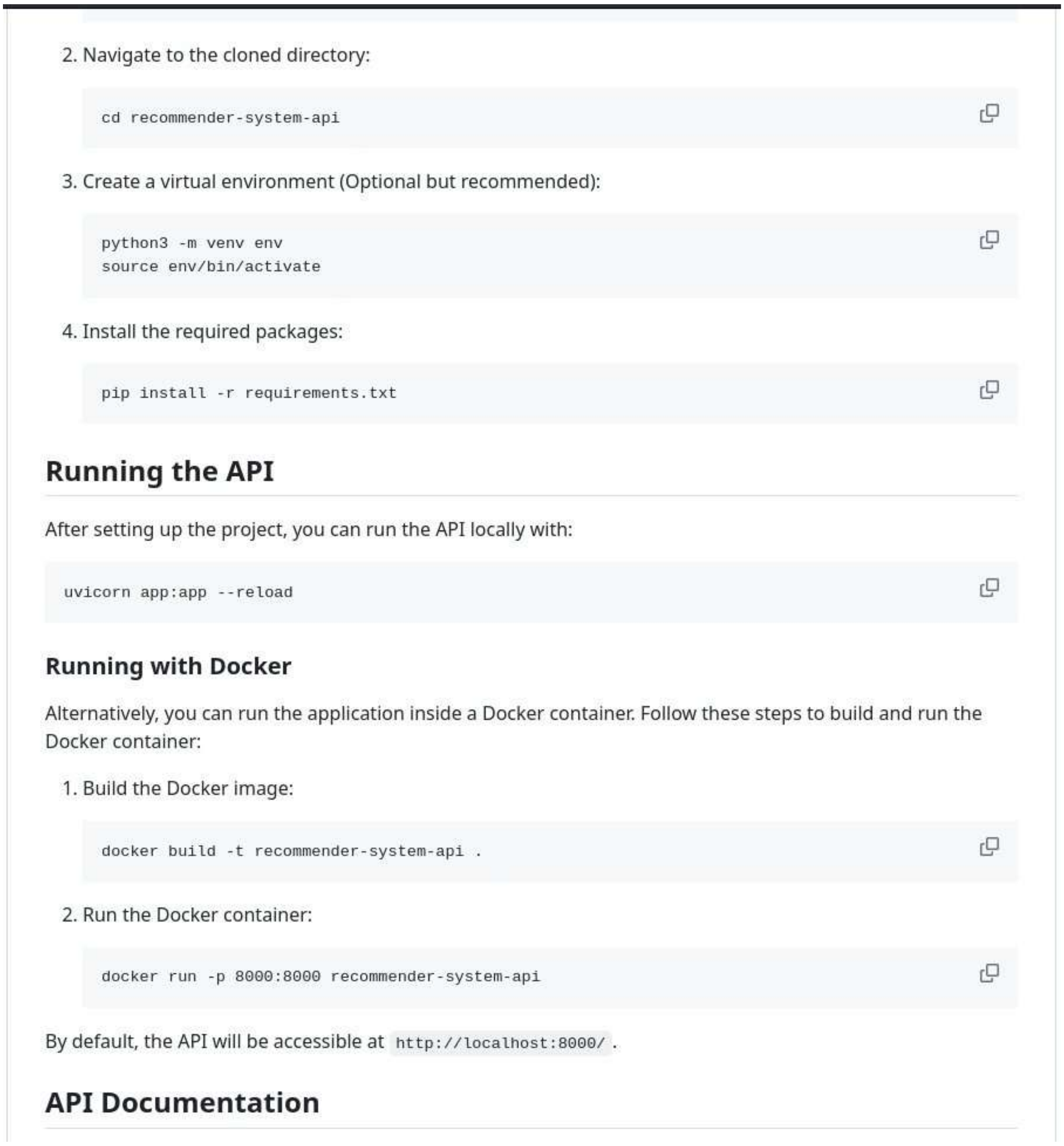
To set up the Recommender System API, follow these steps:

1. Clone the repository:

```
git clone https://github.com/<your username>/recommender-system-api.git
```

2. Navigate to the cloned directory:

```
cd recommender-system-api
```



2. Navigate to the cloned directory:

```
cd recommender-system-api
```

3. Create a virtual environment (Optional but recommended):

```
python3 -m venv env
source env/bin/activate
```

4. Install the required packages:

```
pip install -r requirements.txt
```

Running the API

After setting up the project, you can run the API locally with:

```
uvicorn app:app --reload
```

Running with Docker

Alternatively, you can run the application inside a Docker container. Follow these steps to build and run the Docker container:

1. Build the Docker image:

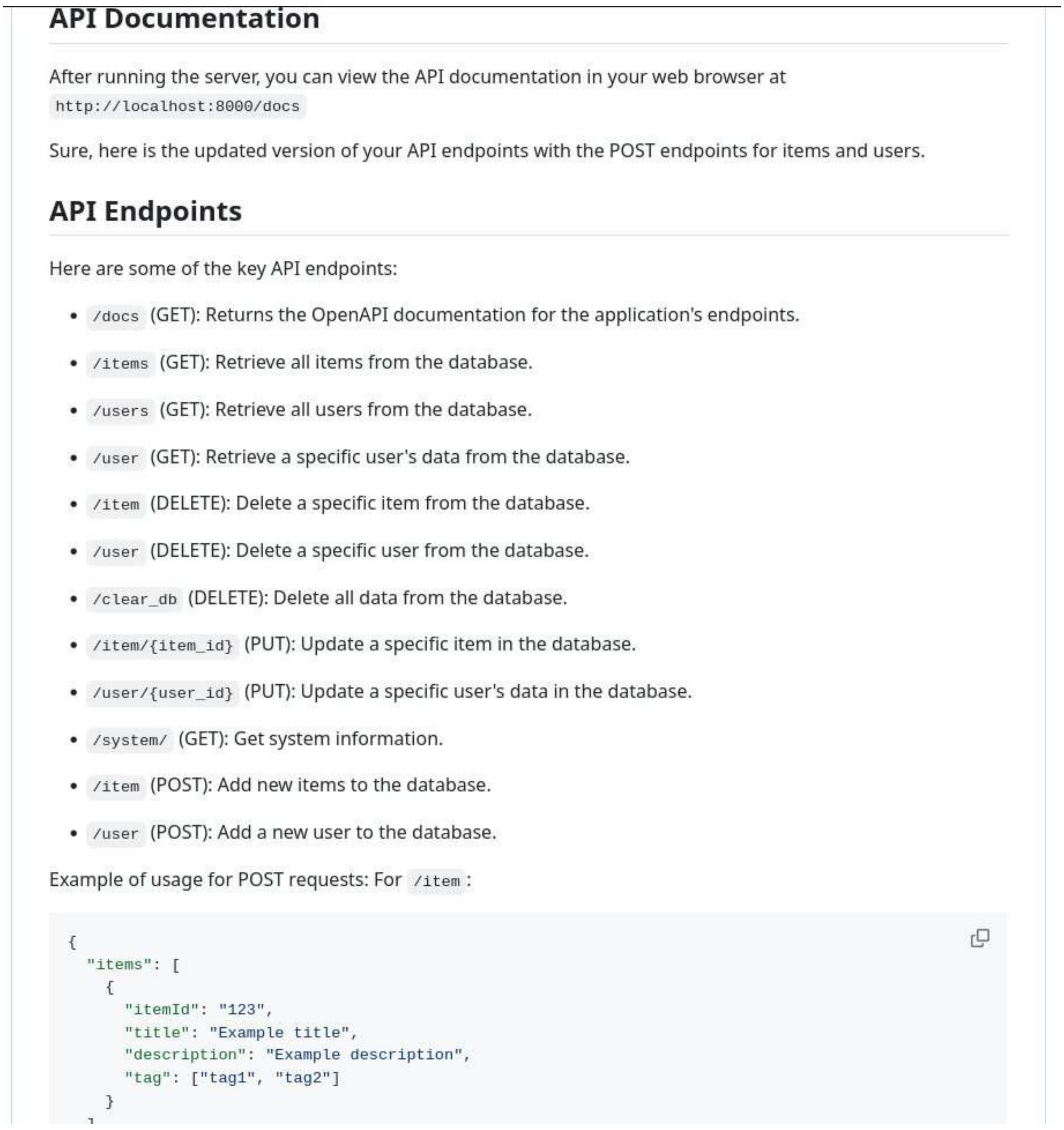
```
docker build -t recommender-system-api .
```

2. Run the Docker container:

```
docker run -p 8000:8000 recommender-system-api
```

By default, the API will be accessible at `http://localhost:8000/`.

API Documentation



API Documentation

After running the server, you can view the API documentation in your web browser at `http://localhost:8000/docs`

Sure, here is the updated version of your API endpoints with the POST endpoints for items and users.

API Endpoints

Here are some of the key API endpoints:

- `/docs` (GET): Returns the OpenAPI documentation for the application's endpoints.
- `/items` (GET): Retrieve all items from the database.
- `/users` (GET): Retrieve all users from the database.
- `/user` (GET): Retrieve a specific user's data from the database.
- `/item` (DELETE): Delete a specific item from the database.
- `/user` (DELETE): Delete a specific user from the database.
- `/clear_db` (DELETE): Delete all data from the database.
- `/item/{item_id}` (PUT): Update a specific item in the database.
- `/user/{user_id}` (PUT): Update a specific user's data in the database.
- `/system/` (GET): Get system information.
- `/item` (POST): Add new items to the database.
- `/user` (POST): Add a new user to the database.

Example of usage for POST requests: For `/item`:

```
{
  "items": [
    {
      "itemId": "123",
      "title": "Example title",
      "description": "Example description",
      "tag": ["tag1", "tag2"]
    }
  ]
```

```
{
  "items": [
    {
      "itemId": "123",
      "title": "Example title",
      "description": "Example description",
      "tag": ["tag1", "tag2"]
    }
  ]
}
```

For `/user`:

```
{
  "items": [
    {
      "userId": "123",
      "itemId": "123",
      "rating": 5,
      "timestamp": "2023-06-13 13:45:30"
    }
  ]
}
```

Remember to replace the example content with your own data.

The API also includes a `/docs` endpoint that provides a user-friendly interface for exploring and testing the API's endpoints. This is an automatically generated API documentation that describes all the endpoints, their methods, parameters, and responses.

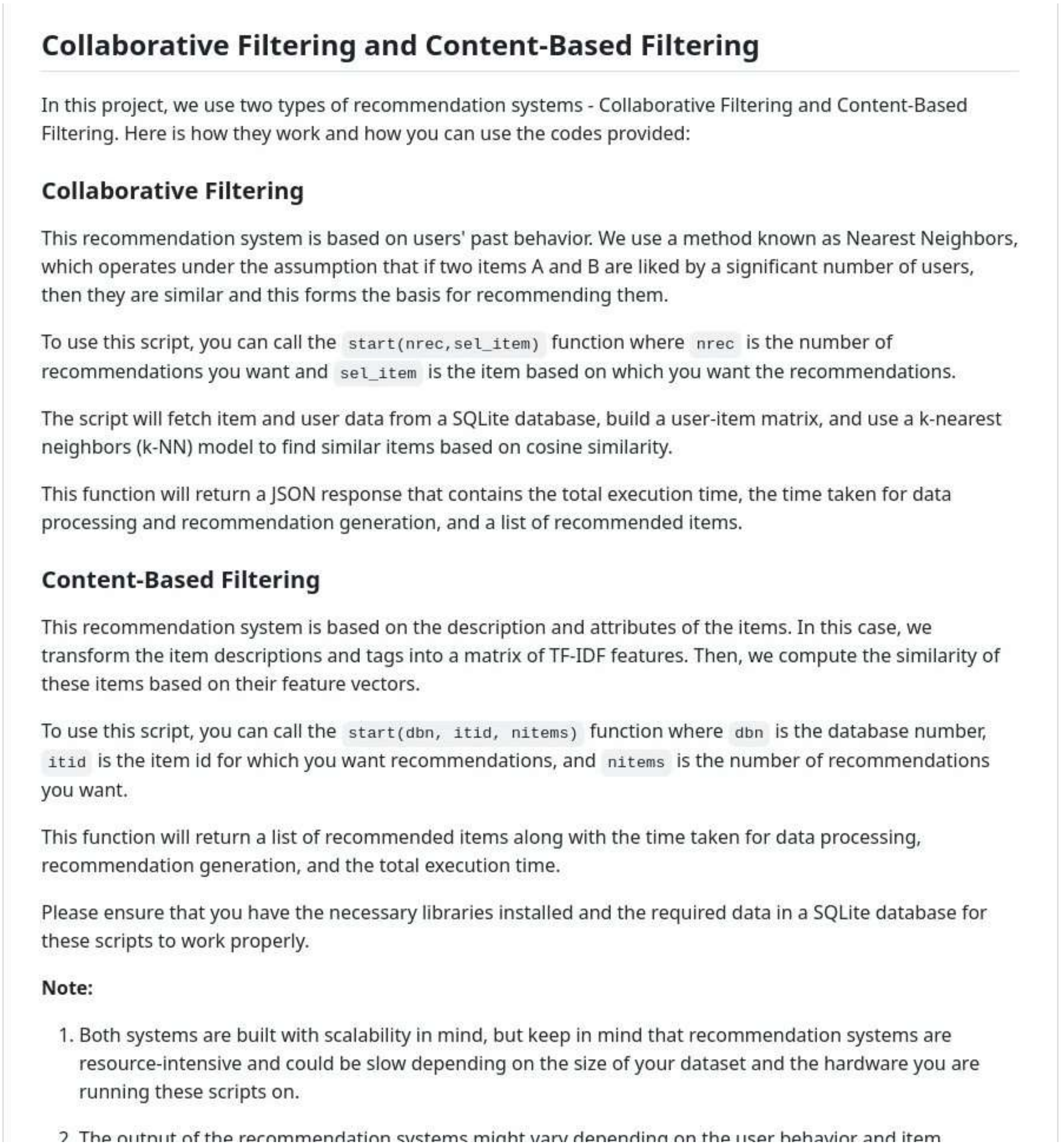
The `/docs` endpoint is accessible by simply appending `/docs` to the API's base URL (e.g., `http://localhost:8000/docs` if the API is running locally on port 8000). This endpoint uses the OpenAPI (formerly Swagger) specifications to generate a comprehensive documentation for the API.

I have also included a postman collection that should make testing easier through the file recsys.postman_collection.json.

Collaborative Filtering and Content-Based Filtering

In this project, we use two types of recommendation systems - Collaborative Filtering and Content-Based Filtering. Here is how they work and how you can use the codes provided:

Collaborative Filtering



Collaborative Filtering and Content-Based Filtering

In this project, we use two types of recommendation systems - Collaborative Filtering and Content-Based Filtering. Here is how they work and how you can use the codes provided:

Collaborative Filtering

This recommendation system is based on users' past behavior. We use a method known as Nearest Neighbors, which operates under the assumption that if two items A and B are liked by a significant number of users, then they are similar and this forms the basis for recommending them.

To use this script, you can call the `start(nrec,sel_item)` function where `nrec` is the number of recommendations you want and `sel_item` is the item based on which you want the recommendations.

The script will fetch item and user data from a SQLite database, build a user-item matrix, and use a k-nearest neighbors (k-NN) model to find similar items based on cosine similarity.

This function will return a JSON response that contains the total execution time, the time taken for data processing and recommendation generation, and a list of recommended items.

Content-Based Filtering

This recommendation system is based on the description and attributes of the items. In this case, we transform the item descriptions and tags into a matrix of TF-IDF features. Then, we compute the similarity of these items based on their feature vectors.

To use this script, you can call the `start(dbn, itid, nitems)` function where `dbn` is the database number, `itid` is the item id for which you want recommendations, and `nitems` is the number of recommendations you want.

This function will return a list of recommended items along with the time taken for data processing, recommendation generation, and the total execution time.

Please ensure that you have the necessary libraries installed and the required data in a SQLite database for these scripts to work properly.

Note:

1. Both systems are built with scalability in mind, but keep in mind that recommendation systems are resource-intensive and could be slow depending on the size of your dataset and the hardware you are running these scripts on.
2. The output of the recommendation systems might vary depending on the user behavior and item

these scripts to work properly.

Note:

1. Both systems are built with scalability in mind, but keep in mind that recommendation systems are resource-intensive and could be slow depending on the size of your dataset and the hardware you are running these scripts on.
2. The output of the recommendation systems might vary depending on the user behavior and item attributes.
3. The quality of recommendations depends on the quality and quantity of data. More data generally leads to better recommendations.
4. The current implementations use basic versions of Collaborative Filtering and Content-Based Filtering. There are more advanced techniques and improvements that can be incorporated to improve the recommendations.

Recsys API Guide

This guide provides instructions on how to interact with the Recsys API.

Clear the Database

Use the following `DELETE` request to clear the database.

```
curl -X DELETE http://localhost:8000/clear_db
```

Add Items

Items can be added to the database with a `POST` request as shown below:

```
curl -X POST http://localhost:8000/item -d @items.json -H "Content-Type: application/json"
```

The `items.json` file should have the following structure:

```
{
    "items": [
        {
            "itemId": "1",
        {
            "itemId": "1",
            "title": "Statue of Liberty",
            "description": "A colossal neoclassical sculpture on Liberty Island in New York Har
            "tag": ["USA", "New York", "Monument"]
        },
        ...
    ]
}
```

Fetch Items

To fetch items from the database, use the following `GET` request:

```
curl -X GET http://localhost:8000/items
```

Add Users

Users can be added to the database with a `POST` request:

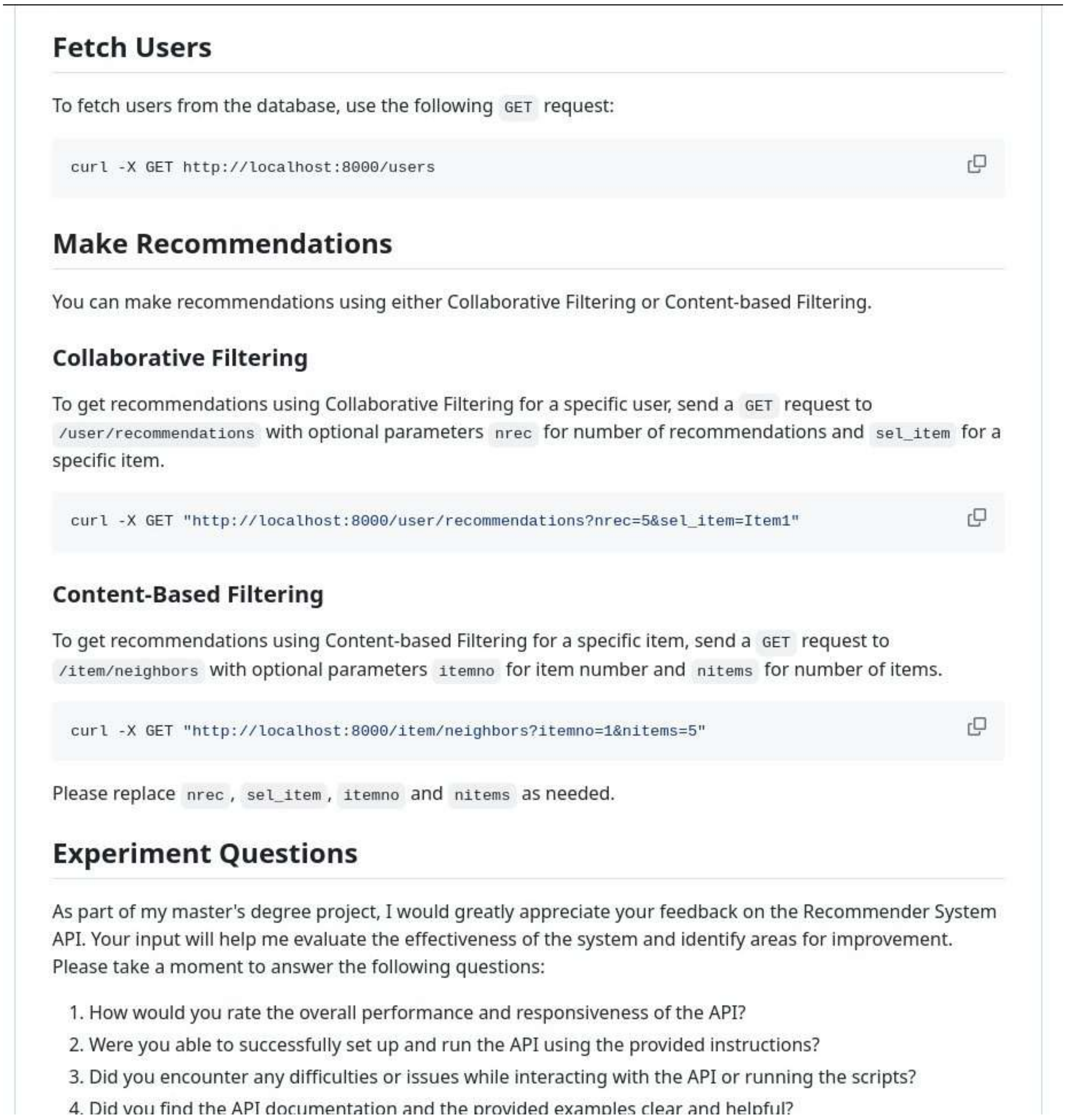
```
curl -X POST http://localhost:8000/user -d @users.json -H "Content-Type: application/json"
```

The `users.json` file should have the following structure:

```
{
    "items": [
        {
            "userId": 1,
            "itemId": 1,
            "rating": 3.5,
            "timestamp": 1112486027
        },
        ...
    ]
}
```

Fetch Users

To fetch users from the database, use the following `GET` request:



Fetch Users

To fetch users from the database, use the following `GET` request:

```
curl -X GET http://localhost:8000/users
```

Make Recommendations

You can make recommendations using either Collaborative Filtering or Content-based Filtering.

Collaborative Filtering

To get recommendations using Collaborative Filtering for a specific user, send a `GET` request to `/user/recommendations` with optional parameters `nrec` for number of recommendations and `sel_item` for a specific item.

```
curl -X GET "http://localhost:8000/user/recommendations?nrec=5&sel_item=Item1"
```

Content-Based Filtering

To get recommendations using Content-based Filtering for a specific item, send a `GET` request to `/item/neighbors` with optional parameters `itemno` for item number and `nitems` for number of items.

```
curl -X GET "http://localhost:8000/item/neighbors?itemno=1&nitems=5"
```

Please replace `nrec`, `sel_item`, `itemno` and `nitems` as needed.

Experiment Questions

As part of my master's degree project, I would greatly appreciate your feedback on the Recommender System API. Your input will help me evaluate the effectiveness of the system and identify areas for improvement. Please take a moment to answer the following questions:

1. How would you rate the overall performance and responsiveness of the API?
2. Were you able to successfully set up and run the API using the provided instructions?
3. Did you encounter any difficulties or issues while interacting with the API or running the scripts?
4. Did you find the API documentation and the provided examples clear and helpful?

5. Were you able to understand and utilize the Collaborative Filtering and Content-Based Filtering recommendation systems effectively?
6. Did the recommendations generated by the system align with your expectations? Were they useful and relevant?
7. Were there any specific features or functionalities that you felt were missing from the API?
8. Do you have any suggestions for improving the API or the recommendation systems?
9. How does the Recommender System API compare to other recommender system APIs you may have used in the past? Please provide your insights and comparisons regarding the features, usability, performance, and any other aspects you consider relevant when comparing the Recommender System API to other similar APIs you have worked with.

Your feedback is invaluable to me, and I appreciate you taking the time to help me with my project. If you have any additional comments or insights, please feel free to share them. Thank you!

Project TODOs

Here is a list of potential improvements and future directions for the project.

- [ ] **Security Integration:** Implement JWT (JSON Web Tokens) for secure transmission of information as a JSON object. This could be used to verify and authenticate requests and manage user sessions.
- [ ] **Database Upgrade:** Replace the current SQLite-based setup with a more robust and scalable database system. Options could include PostgreSQL for relational data or NoSQL alternatives like MongoDB for more flexible data structures.
- [ ] **Hardware Acceleration:** Add support for hardware acceleration (e.g., GPUs, TPUs) to speed up computation of recommendations. This could be achieved through integration with libraries like TensorFlow, PyTorch, or Rapids.
- [ ] **Library Migration (Pandas to Polars):** Migrate from the Pandas library to the newer and more performant Polars library for data manipulation and analysis. This should improve the speed and memory efficiency of data processing operations.

Each of these improvements would contribute significantly to the scalability, performance, and functionality of the project. However, they also each present their own unique challenges and complexities, and should be carefully planned and implemented.

License

This project uses the following license: GPL3 License.

Disclaimer

This project is currently in the development stage and might be subject to changes.