UNIVERSIDADE FEDERAL DE SANTA CATARINA

Hans Buss Heidemann

Projeto e Implementação de uma API REST para Sistemas de Recomendação

Florianópolis

2023

Hans Buss Heidemann

Projeto e Implementação de uma API REST para Sistemas de Recomendação

Trabalho de Conclusão de Curso apresentado ao Curso de Ciências da Computação, do Departamento de Informática e Estatística - Centro Técnologico da Universidade Federal de Santa Catarina, para obtenção de grau de Bacharel em Ciências da Computação.

Orientador: Dr. Roberto Willrich

Florianópolis

2023

Trabalho aprovado. Florianópolis, 2023.

Prof. Dr.
Coordenadora do Curso

Banca Examinadora:

Dr. Roberto Willrich(Orientador)
Universidade Federal de Santa Catarina

Prof. Dr. Elder Rizzon Santos
Universidade Federal de Santa Catarina

Prof. Dr. Renato Fileto
Universidade Federal de Santa Catarina

Prof. Dr. Ronaldo dos Santos Mello
Universidade Federal de Santa Catarina

Florianópolis
2023

Dedico esse trabalho a todos que me apoiaram nesse processo de construção e elaboração desse trabalho.

Agradecimentos

Agradeço a meu pai e minha mãe pela compressão e carinho. Agradeço o professor Roberto Willrich por acreditar no meu trabalho e estimular o desenvolvimento da pesquisa. Agradeço aos membros da banca pela paciência e dedicação na leitura do trabalho. A todos meus amigos que me acompanharam na minha jornada. E a Brenda pelo incentivo e apoio durante esse processo.

"Tudo o que você deve fazer é escrever uma frase verdadeira. Escreva a frase mais verdadeira que você conheça."

(Ernest Hemingway)

Resumo

Atualmente, devido ao montante de informação, produtos e serviços disponibilizados em rede, é fundamental o oferecimento de serviços que realizam certos tipos de filtragem de informações de acordo com as preferências dos usuários. Esta é a função dos Sistemas de Recomendação (SRs), que são capazes de estimar os perfis dos usuários sobre um dado domínio e encaminhar recomendações personalizadas de itens que sejam potencialmente úteis para estes usuários. Muitos sistemas Web são adaptados para uso de um serviço de recomendação específico, preparando os dados de entrada do algoritmo de recomendação e adaptando o resultado para apresentação ao usuário final. Uma alternativa possivelmente mais interessante é o uso de serviços Web de recomendação, facilitando alterações/atualização do serviço de recomendação, sem grandes impactos no sistema Web. Neste sentido, o presente trabalho especifica uma API (*Application Programming Interface*) e desenvolve uma arquitetura de implementação de um serviço Web de recomendação em conformidade com o padrão da API-REST (*API Representational State Transfer*). Através destas API, qualquer sistema Web autenticado pode se registar, atualizar o perfil dos usuários e solicitar recomendações para um dado usuário. Para avaliar a proposta, foi desenvolvido um protótipo prova-de-conceito da API de recomendação, que foi testada a partir de um serviço Web cliente simulado no contexto de recomendações de filmes e utilizando o dataset Movielens. O protótipo implementa três métodos de recomendação, sendo eles por popularidade, baseado em predição de avaliação dos itens pelo usuário, e baseado em conteúdo dos itens. A partir dos testes realizados, foi possível observar o funcionamento adequado do Serviço Web de recomendação proposto neste trabalho.

Palavras-chave: sistemas de recomendação. perfil do usuário. api rest

Abstract

Currently, due to the amount of information, products and services available on the network, it is essential to offer services that perform certain types of information filtering according to users' preferences. This is the function of Recommendation Systems (SRs), which are capable of estimating user profiles on a given domain and forwarding personalized recommendations for items that are potentially useful for these users. Many Web systems are adapted to use a specific recommendation service, preparing the input data for the recommendation algorithm and adapting the result for presentation to the end user. A possibly more interesting alternative is the use of recommendation Web services, facilitating changes/updates of the recommendation service, without major impacts on the Web system. In this sense, the present work specifies an API (*Application Programming Interface*) and developed an implementation architecture for a recommendation web service that conforms to the API-REST standard (*API Representational State Transfer*). Through these APIs, any Web system can register, update users' profiles and request recommendations for a given user. To evaluate the proposal, a proof-of-concept prototype of the recommendation API was developed, which was tested using a simulated client web service in the context of movies recommendations and using the Movielens dataset. The prototype implements three recommendation methods, which are based on popularity, based on prediction of item evaluation by users, and based on item content. From the tests carried out, it was possible to observe the proper functioning of the recommendation Web Service proposed in this work.

Keywords:recommendation systems. user profile. api rest

Sumário

1 INTRODUÇÃO 11
1.1 Objetivos 12
1.1.1 Objetivo Geral 12
1.1.2 Objetivos Específicos 13
1.2 Estrutura do relatório 13

2 FUNDAMENTAÇÃO TEÓRICA 14
2.1 Web Services 14
2.2 Representational State Transfer (REST) 14
2.2.1 Modo de Funcionamento das API REST 15
2.2.1.1 Requisições dos clientes 16
2.2.1.2 Métodos de autenticação 17
2.2.1.3 Respostas do servidor 17
2.2.2 Etapas para a especificação de uma API 17
2.3 Sistemas de Recomendação 19
2.3.1 Técnicas clássicas de recomendação 20
2.3.1.1 Filtragem Colaborativa (FC) 20
2.3.1.2 Filtragem Baseada em Conteúdo (FBC) 22
2.3.1.3 Filtragem Híbrida (FH) 22
2.3.2 Um exemplo de técnica de recomendação 22
2.3.2.1 Perfil do Usuário 23
2.3.2.2 Processo de Agrupamento 25
2.3.2.3 Determinação dos Vizinhos Próximos 25
2.3.2.4 Construção da Recomendação 26

3 FRAMEWORKS DE RECOMENDAÇÃO 27
3.1 Surprise 27
3.2 QREC 28
3.3 LightFM 28
3.4 Cornac 29
3.5 Case Recommender 29
3.6 LensKit 30
3.7 Elliot 30
3.8 Escolha do framework de recomendação 30

4 WEB SERVICE DE RECOMENDAÇÃO: API E ARQUITETURA . 32

4.1 Especificação da API do Web Service de recomendação 32
4.1.1 Conceitos Básicos 32
4.1.1.1 Contexto 32
4.1.1.2 Autenticação 33
4.1.1.3 Registro de Ações 33
4.1.1.4 Métricas 34
4.1.1.5 Registro de Itens 35
4.1.1.6 Algoritmos de Recomendação 35
4.1.1.7 Base de Recomendações 37
4.1.1.8 Adaptação de Modelos 38
4.1.2 Métodos de Criação e Remoção de Contexto 39
4.1.3 Métodos para Registro de Ações 41
4.1.4 Métodos para Registro e Atualização de Itens 43
4.1.5 Remoção de Usuários 47
4.1.6 Método para solicitar recomendações 47
4.2 Arquitetura do WS de Recomendação 49
4.2.1 Componente Web Service 50
4.2.2 Gerenciador de Contexto 50
4.2.3 Atualizador de Contextos 51
4.2.4 Gerenciador de Recomendação 51
4.2.4.1 Atualizador de Recomendações 52
4.2.4.2 Adaptador de Modelos 54
4.2.4.3 Sistema de Recomendação 54
4.3 Considerações Finais 55

5 WEB SERVICE DE RECOMENDAÇÃO: PROVA DE CONCEITO . 56
5.1 Escolha da tecnologia a ser utilizada 56
5.1.1 Django 56
5.1.2 Flask 57
5.1.3 Connexion 57
5.1.4 Falcon 57
5.1.5 FastAPI 58
5.1.6 Hug 58
5.1.7 Seleção do *framework* 58
5.2 Protótipo desenvolvido 59
5.2.1 API 59
5.2.2 Cliente Implementado 59
5.2.3 Gerenciador de Contexto 60
5.2.4 Sistemas de Recomendação implementados 60
5.2.5 Adaptadores de Modelo 61

5.3 Avaliação do Web Service Implementado . . . 61
5.3.1 Avaliação da funcionalidade dos métodos . . . 62
5.3.2 Avaliação da funcionalidade da recomendação . . . 62

6 CONCLUSÃO . . . 64
6.1 Trabalhos futuros . . . 64

REFERÊNCIAS . . . 65

A APÊNDICE 1 . . . 69

B APÊNDICE 2 . . . 79
B.1 Código Fonte . . . 79
B.1.1 Especificação da API proposta . . . 79
B.1.2 Action.py . . . 89
B.1.3 Adaptador$_{m}odelos.py$ . . . 91
B.1.4 App.py . . . 93
B.1.5 Context.py . . . 94
B.1.6 Item.py . . . 95
B.1.7 Recommendation.py . . . 97
B.1.8 Recommendation$_{t}ools.py$ . . . 99
B.2 Arquivos JSON apresentados . . . 105
B.2.1 Registro de Ações . . . 105
B.2.2 Criação de Métrica . . . 105
B.2.3 Exemplo de descriação de filme . . . 105
B.2.4 Exemplo de configuração de recomendadores . . . 107
B.2.5 Exemplo de uma lista de recomendações para um certo usuário . . . 108
B.2.6 Exemplo de uma lista de recomendação para um certo item . . . 109
B.2.7 Exemplo de requisição para o registro de um contexto . . . 110
B.2.8 Exemplo de resposta positiva a requisição de registro de um contexto . . . 111
B.2.9 Exemplo de requisição para deleção de um contexto . . . 111
B.2.10 Exemplo de uma requisição para registro de ações . . . 112
B.2.11 Exemplo de uma requisição para registro de itens . . . 113
B.2.12 Exemplo de uma requisição para remoção de um certo usuário . . . 113
B.2.13 Exemplo de retorno de um pedido de recomendação . . . 114
B.2.14 Exemplo de uma requisição de um pedido de recomendação . . . 114

1 Introdução

Cada vez mais os usuários da Web têm a sua disposição uma enorme quantidade de recursos digitais (como músicas, filmes e documentos em geral), além de serviços e produtos. Neste relatório, o termo item será utilizado para referenciar qualquer um destes recursos, serviços e produtos. Devido ao montante de itens disponíveis para os usuários da Web, a sobrecarga de informação é um problema recorrente, quando a tarefa do usuário em localizar itens relevantes se torna complicada, devido ao grande número de itens recuperados nas buscas na Web (SALLES, 2017).

Sistemas de Recomendação (SR) podem ser utilizados para solucionar este problema de sobrecarga de itens (CHERNEV; BöCKENHOLT; GOODMAN, 2015), tendo como principal objetivo auxiliar no processo de identificação de itens relevantes aos interesses e preferências dos usuários (CAZELLA; NUNES; REATEGUI, 2010). Tais sistemas auxiliam o usuário no processo de localização de itens que sejam relevantes com base de técnicas de filtragem de informação, removendo informações que sejam irrelevantes ou desnecessárias à determinada pessoa.

Atualmente, existem uma diversidade muito grande de técnicas de recomendação (BOBADILLA et al., 2013; AGGARWAL, 2016). Algumas delas são chamadas de clássicas, como a Filtragem Baseada em Conteúdo (FBC), a Filtragem Colaborativa (FC), e as Filtragens Híbridas (FH), que combinam a FBC e a FC para otimizar os resultados. Existem também técnicas mais avançadas (ou experimentais), como as técnicas baseadas em ontologias (TIRYAKI; YüCEBAş, 2023) ou em grafos (GAO et al., 2023).

De maneira geral, um SR possui três operações principais (SALLES, 2017), sendo elas: a construção dos **Perfis dos Usuários**, responsáveis por manter os dados necessários para a recomendação; a **Filtragem de Informação**, que permite selecionar os itens relevantes com base no perfil do usuário; e o **Ranqueamento dos Itens**, necessário para a geração de uma lista ordenada de itens a serem recomendados ao usuário.

Atualmente, existem diversos *frameworks* em código aberto que permitem a implementação de SRs. Estes *frameworks* são soluções que facilitam o reuso de bibliotecas na implementação de SRs, sendo necessária a configuração e a adaptação dos dados dos itens e usuários para o *framework* escolhido. Uma lista não exaustiva de *frameworks* incluem o Surprise(HUG, 2020), o QREC(YU et al., 2021) e o LightFM(KULA, 2015).

Na maior parte dos sistemas Web que se beneficiam de SR, a implementação dos serviços de recomendação ocorre de forma integrada (ou fortemente acoplada) aos demais módulos do sistema e ao seu banco de dados. À medida que a complexidade dessa implementação cresce, suas limitações começam a aparecer, como por exemplo, a alta

dependência entre componentes do sistema, o alto custo necessário para a manutenção, além da pouca flexibilidade e reusabilidade, em que até mesmo uma pequena mudança pode afetar diversos componentes do sistema, limitando a adaptação do mesmo para uso em outros contextos (KAVANAGH, 2018).

A utilização de uma implementação baseada em Serviços Web (WS - *Web Services*) (COSTA; AGUIAR JANDERSON E MAGALHãES, 2013) permite que o SR seja desenvolvido de forma independente do sistema Web que fará uso das recomendações, resultando em diversos benefícios, como por exemplo:

- O SR pode ser utilizado por outros sistemas Web clientes, não sendo necessário a implementação de novos SRs para cada cliente realizar as recomendações. Ou seja, o WS de recomendação pode ser facilmente utilizado para oferecer serviços de recomendação para outros sistemas Web;
- Há um aumento na resiliência, devido ao fato que a implementação do SR ser descentralizada, quando há uma falha no WS há um impacto mínimo sentido pelo cliente (apenas os serviços de recomendação são interrompidos);
- Existe a flexibilidade nas ferramentas e linguagens de programação a serem utilizadas. Com a utilização de WS, os sistemas clientes e o WS podem usar diferentes sistemas operacionais, linguagens e ferramentas e, ao mesmo tempo, conseguirem se comunicar facilmente via a API;
- O desacoplamento do WS de recomendação do sistema cliente substitui o processamento e armazenamento das informações relacionadas à recomendação no lado dos clientes por trocas de informação entre clientes e o WS, permitindo a integração de SRs sem grandes mudanças na estrutura do código do sistema Web cliente. Isso torna possível realizar melhorias ou correções no método de recomendação, sem alterações necessárias no sistema cliente desde que a API permaneça inalterada.

  Já existem propostas na literatura que envolvem a implementação de WS para a utilização de SR,(AÏMEUR et al., 2007). No entanto, essas propostas se baseiam na implementação de um algoritmo de recomendação para WS e não na utilização de um framework já existente para a implementação de algoritmos adicionais.(PANDHARBALE; MOHANTY; JAGADEV, 2020)

1.1 Objetivos

1.1.1 Objetivo Geral

Este trabalho visa projetar e implementar um WS seguindo os padrões da API REST para um sistema de recomendação. Este WS de Recomendação poderá ser usado

por sistemas Web clientes de diferentes tipos, sejam sistemas de repositórios digitais, sistemas de *e-commerce* e outros, que irão utilizar os métodos da API para continua construção dos perfis dos usuários e para a solicitação de recomendações para um dado usuário. Além da especificação da API, este projeto propõe uma arquitetura de implementação do WS de Recomendação proposto. Como forma de avaliação da proposta, foi desenvolvido um protótipo prova-de-conceito, que foi testado usando um cliente simulando recomendações no domínio de filmes.

1.1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são os seguintes:

- Revisão do estado da arte de Sistemas de Recomendações e Serviços Web;
- Especificação de uma arquitetura de implementação do WS de Recomendação;
- Estudo dos trabalhos relacionados que propõem soluções de SRs baseada em REST-FULL ou outras soluções de *Web Services*;
- Levantamento das implementações em código aberto de WS e sistemas de recomendação visando selecionar as tecnologias mais adequadas para o desenvolvimento de um protótipo do *Web Service* de recomendação;
- Seleção dos algoritmos de recomendação mais pertinentes para uso no caso de estudo deste projeto;
- Desenvolvimento de um protótipo prova-de-conceito do WS de recomendação proposto;
- Desenvolver um sistema cliente do WS de Recomendação a ser usado para realização de testes da proposta;
- Realização de avaliações das medidas de desempenho no estudo de caso desenvolvido.

1.2 Estrutura do relatório

O restante deste relatório é organizado da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica com a revisão dos principais conceitos relacionados a *Web Services*, que serão utilizados no contexto deste projeto, além dos principais conceitos relacionados a Sistemas de Recomendação. O capítulo 3 apresenta um levantamento dos principais *frameworks* de recomendação candidatos para uso neste projeto. Em seguida, o Capítulo 4 apresenta a API e a arquitetura de um WS de recomendação proposto neste projeto. Na sequência, o Capítulo 5 apresenta o protótipo desenvolvido e a avaliação da proposta. Finalmente, o Capítulo 6 apresenta as conclusões e trabalhos futuros.

2 Fundamentação Teórica

2.1 Web Services

Como definido pelo W3C *Web Services Architecture Working Group*, um Serviço Web (WS - *Web service*) é uma aplicação de software identificada por uma URI, nas quais as interfaces são capazes de serem definidas, descritas e descobertas como artefatos XML (FERRIS; FARRELL, 2003). Tais aplicações permitem a interação direta com outros sistemas Web utilizando mensagens baseadas em XML e/ou JSON, trocadas via protocolos de Internet, normalmente utilizando HTTP/HTTPS, mas não restrito a tais, podendo serem utilizados outros mecanismo de transporte, como o FTP.

As vantagens na utilização de WS incluem a desacoplação de interface do serviço da implementação, a vinculação dinâmica de serviços, além do aumento da interoperabilidade multilinguagem e multi-plataforma.

2.2 Representational State Transfer (REST)

Para (FIELDING, 2000), o REST é um estilo arquitetural para sistemas de hipermídia distribuídos criado para guiar o projeto e o desenvolvimento da arquitetura da Web. Esse estilo ignora os detalhes dos componentes da implementação e da sintaxe do protocolo ao propor um conjunto de restrições de como a arquitetura de sistemas de hipermídia distribuídos, como a Web, devem funcionar.

O estilo REST define 6 restrições relacionadas a estilos arquiteturais que são descritos a seguir:

- **Cliente-Servidor**: restrição responsável pela separação dos conceitos de usuário e de armazenamento de dados. Esta restrição aumenta a portabilidade da interface em multi-plataformas e simplifica os componentes do servidor (FIELDING, 2000);
- **Protocolo sem estado**: restrição para as interações cliente-servidor, que não devem manter estado por natureza. Toda a informação necessária para se entender uma requisição deve estar presente na própria requisição;
- **Cache**: restrição que exige que dados contidos em uma requisição devam ser marcados explicitamente ou implicitamente como cacheáveis ou não, ou seja, é marcado

se os dados da requisição podem ser armazenados em cache e utilizados posteriormente. Uma requisição cacheável permite que seus dados possam ser reutilizados pelo cliente do WS sem a necessidade de ser feito um novo pedido ao servidor, aumentando a eficiência, escalabilidade e a performance percebida pelo cliente do WS;

- **Interface Uniforme**: esta restrição é a que diferencia o REST de outras abstrações para a Web, devido a sua ênfase na interface uniforme entre componentes. De fato, existem quatro restrições de interface, que são (FIELDING, 2000):
  - A identificação de recursos em requisições: Recursos individuais são identificados dentro da requisição utilizando URIs;
  - A manipulação de recursos através de representações: A representação de um recurso através de uma URI é suficiente para que um cliente possa deletar ou modificar esse recurso;
  - Mensagens auto-descritíveis: Cada mensagem deve fornecer informação suficiente de como processá-la;
  - Hipermídia como o motor do estado da aplicação: Através do acesso de uma URI inicial, o cliente deve conseguir descobrir e acessar todos os recursos disponíveis.
- **Sistema em camadas**: restrição que define que uma arquitetura deve ser composta de camadas hierárquicas, restringindo o comportamento dos componentes para que cada um não possa ver além da camada intermediária com a qual está interagindo;
- **Code-On-Demand**: a última restrição permite que as funcionalidades do sistema sejam estendidas através do download e execução de código em formas de *applets* e *scripts*. Essa é uma restrição opcional, pois embora melhore a extensibilidade do sistema, ela reduz a visibilidade do sistema por flexibilizar que o servidor que decida como certas coisas serão feitas. Outra preocupação é que essa restrição possibilita que sejam enviados códigos maliciosos para serem executados pelo servidor.

Uma *Interface de Programação de Aplicações* (*API*) que segue todas as restrições REST é chamada de API RESTful.

2.2.1 Modo de Funcionamento das API REST

Embora o uso de uma API REST seja diferenciada para cada serviço, com os desenvolvedores publicando documentações da forma que as APIs devem ser utilizadas, as chamadas para um método da API REST em geral envolvem as seguintes etapas(O. . . ,):

1. Cliente envia uma requisição seguindo os padrões da API disponibilizada pelo lado servidor.
2. Servidor autentifica e verifica se o cliente tem direito a realizar a requisição.
3. Servidor processa a requisição internamente após a recebê-la.
4. Servidor retorna uma resposta ao cliente com informações sobre o sucesso (ou erro) da requisição, além de informações solicitadas.

As seções que seguem descrevem mais em detalhes cada uma destas etapas.

2.2.1.1 Requisições dos clientes

As requisições realizadas pelos clientes de um WS devem atender aos requisitos descritos a seguir:

- **Identificador de recurso exclusivo**: o URL (*Uniform Resource Locator*) da requisição especifica o caminho para o recurso solicitado e é utilizado pelo servidor para a identificação de tal recurso. Este URL é conhecido também como *endpoint* de solicitação;
- **Método**: os clientes devem usar um dos métodos HTTP para informar a ação que o servidor deve realizar sobre o recurso indicado no URL. Os métodos mais comuns são:
  - **GET**: utilizado para acessar o recurso identificado no URL especificado pela requisição;
  - **POST**: tem a mesma função do método GET, mas permite ao cliente enviar dados ao servidor junto com a requisição do recurso;
  - **PUT**: método usado principalmente para atualizar um recurso já existente. Se aquele recurso não existir, a API pode decidir se ele será criado ou não no servidor;
  - **DELETE**: método usado para remover recursos do sistema, atendido caso o cliente possua a autorização necessária;
  - **PATCH**: método utilizado para aplicar modificações parciais em um recurso do sistema. É considerado um conjunto de instruções de como modificar o recurso.
- **Cabeçalhos HTTP**: é formado por um conjunto de metadados trocados entre o cliente e o WS informando o formato da solicitação e da resposta. Além disso, um cabeçalho pode incluir dados para os métodos HTTP operarem corretamente e

parâmetros adicionais que permitam ao WS ter mais clareza sobre o que precisa ser feito (FIELDING, 2000).

2.2.1.2 Métodos de autenticação

Os clientes de um WS REST devem comprovar sua identidade ao servidor para o estabelecimento da confiança. Para isto, existem alguns métodos de autenticação que podem ser utilizados. Os principais métodos de autenticação são (FIELDING, 2000):

- **Autenticação de HTTP**: existem dois esquemas utilizados pelo HTTP para autenticação. A autenticação básica utiliza o nome e a senha do usuário codificados com base64. Já a autenticação do portador utiliza um *token*, normalmente uma cadeia de caracteres criptografada, gerada pelo servidor em resposta a uma solicitação de *login*.
- **Chaves de API**: as chaves de API são geradas pelo servidor a um cliente sempre que o mesmo tenta acessar algum recurso. Essa chave é utilizada pelo cliente para prover sua autenticação e assim ter acesso aos recursos. Este método é menos seguro, pois o cliente precisa transmitir a chave, tornando vulnerável a roubo de rede.
- **OAuth**: é utilizado uma combinação de senhas e *tokens*, em que primeiro é solicitado uma senha e depois um *token* para ser concluída a autenticação. O *token* pode ser verificado a qualquer momento com escopo e longevidade definidos pelo WS.

2.2.1.3 Respostas do servidor

Uma resposta do WS RESTful deve possuir os seguintes itens:

- **Linha de status**: contém um código de status representado por três dígitos que indicam o sucesso ou a falha da solicitação;
- **Cabeçalhos**: contém mais informações sobre a resposta e o servidor, fornecendo a codificação, a data e o tipo do conteúdo;
- **Corpo da mensagem**: contém a representação do recurso utilizando um formato apropriado com base na solicitação do cliente. Os recursos podem ser solicitados em formatos como XML ou JSON.

2.2.2 Etapas para a especificação de uma API

A especificação de uma API, realizada de forma correta, é extremamente importante para que a mesma seja melhor utilizada, além de facilitar para que ela seja adotada por novos desenvolvedores (HENRIQUE et al., 2022). As principais formas de documentação de APIs REST são as seguintes:

- RAML(BAELDUNG, 2022) (*RESTful API Modeling Language*) é uma linguagem baseada em YAML (*Yet Another —Markup Language*) para fazer a descrição de APIs estáticas. Embora ela tenha sido definida para uso na especificação de APIs RESTful, a RAML não consegue descrever as APIs que seguem todas as restrições REST;
- API Blueprint(LTD, 2021) é uma linguagem de descrição de APIs com base em JSON e YAML que tem um maior enfoque na documentação da API;
- Swagger(Wordnik, 2012) é um conjunto de softwares utilizados para construção, documentação, testagem e utilização de *Web services* RESTful baseado em *Markdown* e *Markdown Syntax for Object Notation* (MSON).

Para a explicação de como especificar uma API neste projeto é utilizada o Swagger, tecnologia que utiliza os padrões de OpenAPI.

A especificação da API pode ser feita em ferramentas como o *Swagger Editor*(Wordnik, 2012), que permite que definições de API OpenAPI possam ser alteradas diretamente no navegador e a documentação possa ser gerada em tempo real ou apenas utilizando editores de texto.

Uma API definida utilizando uma especificação OpenAPI pode ser descrita em três seções centrais:

- Meta informação
- Itens do *Path*:
  - Parâmetros
  - Corpos do pedido
  - Respostas
- Componentes reusáveis:
  - *Schemas* (Modelos de dados)
  - Parâmetros
  - Respostas
  - Outros componentes

O primeiro passo é a definição das metainformações da API, em que serão descritos e identificados o tipo e versão da especificação, além de outras informações legíveis para humanos, como título, descrição e versão da API, os quais são especificadas utilizando a *keyword info*.

Após isso é necessário que seja especificado qual URL do servidor será utilizada para as chamadas da API, podendo também serem declarados na seção de componentes uma autenticação básica para segurança da implementação.

O próximo passo é a inclusão dos recursos, em que cada recurso é declarado através de um *endpoint* relativo à URL do servidor. Junto da definição dos *endpoints*, é necessário definir os métodos HTTP que poderão utilizar os recursos dos mesmos.

Para cada método HTTP, é necessário que sejam definidas as respostas que o mesmos podem retornar. Cada resposta deve conter pelo menos um código de *status* HTTP, além de uma descrição que detalhe a resposta.

Existem três tipos de parâmetros que devem serem declarados:

- *Query Params*: são o tipo mais comum de parâmetros. Aparecem no final da URL e são opcionais, podem ser declarados múltiplas vezes na URL e são um componente não-hierárquico. A definição do mesmo deve ser realizada abaixo do objeto de parâmetros, local que deve conter informações sobre em que deve estar presente o parâmetro, nesse caso, na *query*, além da descrição do mesmo e qual o tipo de dados que é recebido;
- *Body Params*: são tipicamente utilizados em operações POST, PUT e PATCH. Para sua declaração é necessário utilizar o objeto *requestBody*, que define os itens obrigatórios que devem estar presentes no parâmetro, além de outras propriedades;
- *Path Params*: podem ser utilizados para isolar um componente específico dos dados que estão sendo trabalhados. Esses parâmetros são declarados no final da URL, mas são um componente hierárquico e empilhados sequencialmente. Para esse tipo de parâmetros, é necessário que sejam declarados novos *endpoint* com o parâmetro presente na URL. Eles são declarados abaixo do objeto de parâmetros.

Para facilitar a especificação de uma API, existem diversos tipos de componentes que podem ser reutilizados. Para que isso seja possível, é necessário que o mesmo seja especificado abaixo da seção responsável pelos componentes reutilizáveis, chamada *components*, e que seja adicionado na seção de componentes do método que utilizará o componente, adicionando a seguinte linha "$ref: '#/components/component/name'", em que *component* e *name* são respectivamente o tipo e o nome do componente.

2.3 Sistemas de Recomendação

Sistemas de recomendação (SR) estão sendo cada vez mais utilizados na Web, seja no momento da busca por um filme ou um item em um site de *e-commerce*, ou mesmo na hora de receber um *post* de um conhecido em uma rede social. SRs estão presentes nesses

momentos com o objetivo apoiar a localização de itens que sejam do interesse do usuário em meio à quantidade gigantesca de itens que são disponíveis na Web. Desta forma, os SRs auxiliam em minimizar o problema chamado de *Sobrecarga de itens* (SALLES, 2017).

Existem várias definições para SRs, que variam dependendo do autor que se consulta. Para (RESNICK; VARIAN, 1997), um SR tem por objetivo apoiar o processo social de indicar ou receber recomendações, no qual se busca a identificação de um conjunto de itens com base nos interesses do usuário. Outros autores, como (RUOTSALO, 2010), definem SR como um tipo especifico de sistemas de filtragem de informação, utilizados para a identificação de um conjunto de itens relevantes ao usuário.

Este capítulo tem por objetivo apresentar uma visão geral dos SRs, incluindo as técnicas clássicas de recomendação, em que serão abordadas as vantagens e desvantagens de cada técnica.

2.3.1 Técnicas clássicas de recomendação

Diversas abordagens podem ser utilizadas para realizar a filtragem de informação para a recomendação. Nesta seção são apresentadas as três técnicas clássicas de recomendação, a saber: Filtragem Colaborativa (FC), Filtragem Baseada em Conteúdo (FBC) e Filtragem Híbrida.

2.3.1.1 Filtragem Colaborativa (FC)

A FC é uma técnica de filtragem que pressupõe que existe uma grande probabilidade de que um usuário goste dos itens que atendem as preferências de outros usuários com perfis similares ao seu (SALLES, 2017). Desta forma, a FC busca identificar grupos de usuários com perfis semelhantes, os chamados de **Vizinhos Próximos**, e, a partir destes, realizar a recomendação com base na estimativa de avaliações (*ratings*) que o usuário foco da recomendação poderia atribuir aos itens.

Existem duas categorias em que algoritmos de FC podem ser divididos (BOUTEMEDJET; ZIOU, 2008):

Algoritmos baseados em memória: em que a similaridade dos perfis de usuários é calculada comparando as avaliações dos itens a recomendar feitas por estes usuários. Para determinar a similaridade entre os perfis de usuários, é utilizada uma das diversas métricas de similaridade disponíveis atualmente, como o Coeficiente de Correlação de Pearson e a Similaridade por cosseno (CAZELLA; NUNES; REATEGUI, 2010). Estas avaliações podem ser explícitas, em que o usuário envia explicitamente uma avaliação do item, por exemplo, atribuindo uma nota de 1 a 5. As avaliações implícitas são aquelas que são estimadas pelo SR, sem a intervenção di-

reta do usuário. Um exemplo de avaliação implícita é considerar a compra de itens como indicação de que o usuário tem preferência pelo item (e suas características);

Algoritmos baseados em modelos: utilizam modelos estatísticos construídos pelo algoritmo para a estimação das preferências dos usuários. Tais modelos podem utilizar redes Bayesianas, redes de dependência e diagnóstico de popularidade.

Em relação aos algoritmos de FC baseados em memória, existem duas abordagens que são matematicamente similares, mas conceitualmente diferentes:

- FC baseada em Usuário: esta abordagem foi descrita anteriormente, em que o sistema determina usuários similares (vizinhos próximos) de um usuário u foco da recomendação, e estima uma avaliação de um item que o usuário u atribuiria com base nas avaliações dadas ao item pelos seus vizinhos próximos;

- FC baseada em Item: nesta abordagem, para um dado item i, o sistema determina um conjunto de itens similares a i e a avaliação de um usuário u é estimada com base nas suas avaliações atribuídas a itens similares a i.

Vários algoritmos de FC são baseados em modelos de fatoração de matrizes (KOREN; BELL; VOLINSKY, 2009), em que as linhas da matriz contém os itens e as colunas as avaliações dos usuários. A fatoração de matrizes permite determinar característica latentes, permitindo identificar relações entre usuários e itens e predizer como os itens poderiam ser avaliados pelos usuários. Existem diversas técnicas baseadas em fatoração de matrizes, como a SVD (*Singular Value Decomposition*)(SARWAR et al., 2000), a PMF (*Probabilistic Matrix Factorization*(SALAKHUTDINOV; MNIH, 2007), a SVD++(RICCI; ROKACH; SHAPIRA, 2010), a NMF (*Non-negative Matrix Factorization*)(LUO et al., 2014) e várias outras.

As técnicas de FC possuem algumas limitações bem conhecidas, como apontadas por (CLAYPOOL et al., 1999). Um dos problemas é o chamado de partida a frio (*(Cold-Start)*), que ocorre no momento inicial de operação de um sistema, quando existe pouca informação sobre os usuários, dificultando a localização de seus vizinhos próximos e impossibilitando assim as recomendações. Outro problema conhecido da FC é o da esparsidade, situação em que existem itens no sistema que não foram avaliados ainda ou que poucos usuários o fizeram. Assim, estes itens novos ou impopulares dificilmente serão recomendados.

Outro problema é a técnica não classificar as preferências de um usuário por sua opinião própria (seu perfil), mas estimá-la a partir de outros com opiniões parecidas. Assim a preferência de um nem sempre reflete a de outro (CLAYPOOL et al., 1999)(PARK et al., 2012).

2.3.1.2 Filtragem Baseada em Conteúdo (FBC)

Na FBC, a recomendação de itens é realizada a partir da comparação entre o perfil do usuário foco da recomendação e as características associadas ao item, como metadados ou seu próprio conteúdo. Para isso, a FBC observa o comportamento do usuário e, a partir disso, recomenda itens que tenham relação com seu perfil, em que são definidas as preferencias e os interesses do usuário com base nas características dos itens que o mesmo gostou, podendo terem sido avaliados ou não (SALLES, 2017).

Para (BURKE, 2007), as recomendações geradas pela FBC podem utilizar duas fontes de dados: as características associadas aos itens, seu próprio conteúdo ou os metadados que o descrevem, e as avaliações dos itens realizadas pelo usuário.

Devido ao fato da FBC não considerar o perfil de outros usuários para a recomendação, ela possui como vantagem em relação à FC o quesito da privacidade dos dados do usuário(MOONEY; ROY, 2000), pois não deixa evidente o que outros usuários têm acessado, fato que acontece com a FC ao levar em consideração o perfil de seus usuários semelhantes.

A FBC também sofre do problema de Partida a Frio, em que a realização de recomendações a usuários recentes ou que utilizam pouco o sistema é dificultada devido aos mesmos possuírem um histórico de limitação muito pequeno, com um perfil pouco preciso. Outro problema é o *Fechamento dos itens a serem recomendados*, o qual se dá pela recomendação de itens similares em conteúdo a itens que já foram avaliados ou acessados pelo usuário, impedindo a recomendação de itens que possam ser relacionados, mas tenham conteúdo diferente.

2.3.1.3 Filtragem Híbrida (FH)

Embora as técnicas de FC e FBC sejam muito utilizadas, elas ainda apresentam desvantagens. Para tentar mitigar estes problemas, foram propostas diversas técnicas que combinam a FC e a FBC, resultando nas chamadas técnicas híbridas. As técnicas hibridas de recomendação combinam as vantagens de ambas as técnicas e geralmente tentam lidar com o problema da partido a frio.

A recomendação na FH é geralmente gerada utilizando tanto o perfil do usuário em foco (FBC), como o perfil de outros usuários similares (FC). Segundo (MOONEY; ROY, 2000), as técnicas FBC e FC são complementares e juntando-as pode se conseguir resultados mais satisfatórios para a recomendação.(SALLES, 2017)

2.3.2 Um exemplo de técnica de recomendação

Esta seção apresenta a técnica de recomendação proposta por (CASAGRANDE, 2014), baseada na Filtragem Colaborativa (FC) e que pode ser aplicada a qualquer sistema

em que o conteúdo disponibilizado é descrito por um conjunto de metadados modelados pelas regras que serão descritas.

A técnica assume a hipótese que um usuário individual pertencente a um grupo de perfis semelhantes tem grande probabilidade de gostar de conteúdos acessados por usuários desse mesmo grupo e a técnica é divida em quatro etapas: construção dos perfis do usuário; processo de agrupamento de usuários; determinação de vizinhos próximos em cada grupo; e construção da recomendação (CASAGRANDE, 2014).

2.3.2.1 Perfil do Usuário

Cada usuário cadastrado no sistema é associado a um perfil que permite a captura das preferências e interesses do mesmo. (FURTADO et al., 2009) propõe um modelo que permite determinar o perfil do usuário de maneira implícita, sem a intervenção do usuário e através da analise da frequência de ocorrência de valores dos elementos de metadados que ocorrem nos conteúdos acessados.

A definição de Biblioteca Digital utilizada por esta técnica sendo requirida por (FURTADO et al., 2009) pode ser definida como:

$$RD = (D, M, U, PU) \tag{2.3.1}$$

Em que:

- $D = \{d|i \in [1, I]\}$ definindo o conjunto de documentos que compõem a coleção do sistema, com um total de I itens.
- $M = \{md_j|j \in [1, J]\}$ Define o conjunto de elementos de metadados utilizados que descrevem documentos do sistema. J se refere ao número de elementos de metadados e $d_i.md_j$ se refere ao valor do elemento de metadado md_j do documento d_i
- $U = \{u_k|k \in [1, K]\}$ sendo o conjunto de usuários do sistema, com K sendo o número de usuários registrados.
- $PU = \{p_k|k \in [1, K]\}$ sendo o conjunto de perfis de usuários, com cada usuário u_k associado a um perfil p_k

Para cada usuário u_k, é associado um perfil pu_k definido pela seguinte fórmula:

$$pu_k = (PE_k, CA_k)$$

Em que PE_k são as Preferências Específicas do usuário relacionadas ao domínio do sistema e capturadas de forma implícita através de uma análise de frequência dos valores de um subconjunto dos elementos de metadados e CA_k são os Conteúdos Acessados pelo usuário, sugerindo que o mesmo se interessa pelo conteúdo.

Para se determinar o conjunto de elementos de metadados utilizados para a obtenção das preferências específicas, chamado de Metadados Observados (MO), é necessário analisar quais metadados são relevantes na determinação das preferências, sendo elas dependentes do domínio do sistema. Os MOs são definidos por:

$$MO = \{mo_j | j \in [1, J]\} \tag{2.3.2}$$

Em que MO é o subconjunto de elementos de metadados para a construção do perfil do usuário e J é o número de elementos de metadados observados.

O conjunto de documentos acessados pelo usuário é definido por:

$$CA = \{d_k | k \in [0, K_j]\} \tag{2.3.3}$$

Sendo d_k o conjunto de documentos acessados pelo usuário i e K_j é o número total de conteúdos que esse usuário acessou.

As preferências específicas de um usuário são definidas por (FURTADO et al., 2009):

$$PE = \{SV)_j | j \in [1, J]\} \tag{2.3.4}$$

Sendo o conjunto de valores pem querados do elemento de metadado observado mo_j, construído a partir de acessos do usuário u_j e definido por:

$$SV_j = \{vw_{n'} | n \in [1, N_j]\} \tag{2.3.5}$$

Em que vw_n é o valor pem querado (v_n, w_n), com v_n sendo o valor do elemento do metadado mo_j observado e w_n representando o peso da preferência de valor v_n. Já N_j é o número de valores distintos do elemento de metadado mo_j.

Sempre que um usuário acessa um novo documento são atualizados os conjuntos de valores pem querados dos elementos de metadado observado. Para isso existem três análises básicas utilizadas para incrementar esses pesos de preferências(CHEN; LAPAUGH; SINGH, 2002):

- **Análise de Existência**: a preferência pode assumir um peso de 0 ou 1. Se o usuário acessou documentos com esse valor de MO, o peso será 0, caso contrário será 1;
- **Análise de Frequência**: o peso é definido pela frequência de ocorrências de valores de MO no documento acessado. Para isto existem dois métodos que estimam o peso da preferencia: *Past-Days* (PD) são utilizados somente os valores encontrados nos documentos acessados nos últimos n dias pelo usuário; e *Whole-History* define o peso através do número total de acessos;
- **Análise de Idade de Acesso**: é considerado que acessos mais antigos do usuário são menos relevantes para considerar a preferência do mesmo do que artigos que foram acessados mais recentemente.

2.3.2.2 Processo de Agrupamento

Através da análise dos perfis dos usuários é realizado o processo de agrupamento, em que um usuário é incluído em um grupo quando um determinado valor de MO atingir o *limiar de agrupamento*.

Este conjunto de agrupamento de usuários pode ser definido por:

$$G = \{g_{j,v_n} | j \in [1, J], n \in [1, N_j\} \tag{2.3.6}$$

em que J é o número de MO considerado no agrupamento e N_j é o número de valores distintos do metadado j.

O Algoritmo 1 a seguir especifica a técnica de agrupamento. O algoritmo inicia com o conjunto de grupo G vazio e em seguida analisa o peso dos valores dos metadados observados de cada perfil de usuário. Se o valor de um metadado atingir a limiar de agrupamento, o usuário é incluído no grupo identificado.

Algoritmo 1 Algoritmo de agrupamento adotado

```
G ← {}
para todos todos os usuários u_i do sistema faça
  para todos todo SV_j em PE_i faça
    para todos todo vw_u em SV_j faça
      se W_n ≥ limiarAgrupamento_j então
        se grupo g_{j,vn} não existir então
          Inclua este grupo em G
        fim se
        Inclua o usuário u_i no grupo g_{j,vn}
      fim se
    fim para
  fim para
fim para
```

2.3.2.3 Determinação dos Vizinhos Próximos

O próximo passo a ser realizado é a identificação dos "vizinhos próximos"de um usuário Q. Para isso, é calculada em duas etapas a similaridade entre o perfil de tal usuário e o perfil dos outros usuários que pertencem a um mesmo grupo que Q. Primeiro se determina a similaridade entre os conjuntos de valores ponderados de cada um dos valores dos metadados observados (SVs). Então se determina a similaridade de perfis de usuário.

A similaridade entre os conjuntos SVs de um metadado observado MO_i é determinada utilizando a função do cosseno, fornecendo resultados entre 0 (nenhuma relação entre os perfis) e +1 (perfis idênticos). A similaridade entre os conjuntos SVs de um

metadado observado é determinada por:

$$similaridade\_cosseno_{mo_i}(Q,D) = \frac{\sum_{n=1}^{N} W_{qn} \cdot W_{dn}}{\sqrt{\sum_{n=1}^{N}(W_{qn})^2 \cdot \sum_{N}^{n=1}(W_{dn})^2}} \tag{2.3.7}$$

Em que:

- Q representa o usuário foco da recomendação;
- D representa outro usuário a ser comparado;
- N é o número de valores diferentes do elemento de metadado mo_i definidos no conjunto SV_i do perfil do usuário Q;
- W_{qn} é o peso de preferência do valor v_{in} do elemento de metadados mo_i.

Nesta proposta, os vizinhos próximos de um usuário Q são todos aqueles que tem uma similaridade de perfil com o usuário Q acima do limiar de similaridade. Esse valor depende do domínio do sistema e dos metadados observados selecionados. Os vizinhos podem ser definidos da seguinte maneira:

$$Vizinhos_Q = \{D | similaridade_{perfil}(Q,D) \geq limiar\} \tag{2.3.8}$$

2.3.2.4 Construção da Recomendação

A etapa final tem como objetivo produzir a lista ordenada de conteúdos que serão recomendados ao usuário foco. A lista é construída a partir dos conteúdos acessados pelos vizinhos próximos desse usuário e que ainda não foram acessados pelo mesmo.

O ordenamento da lista é feita com base no cálculo de peso de preferência definido por (WILLRICH et al., 2006), desta forma os conteúdos que devem aparecer primeiro são aqueles que tiverem maior chance de serem interessantes para o usuário. A lista ordenada é definida por:

$$CR = (d_1, d_2, ..., d_R) emque \forall r \in [1, R-1], w_{pref}(u_k, d_r) \geq W_{pref}(u_k, d_{r+1}) \tag{2.3.9}$$

3 *frameworks* de Recomendação

Como colocado em (SALLES, 2017), *frameworks* são soluções que facilitam o reuso de bibliotecas na implementação de sistemas da recomendação, sendo necessária a configuração e adaptação dos dados dos itens e usuários para o *framework* em questão. Esse capítulo busca apresentar alguns dos principais *frameworks* existentes. Para isto, foi realizado um levantamento de *frameworks* citados em artigos científicos, além de pesquisar os disponíveis no site **Github**.

Os critérios escolhidos para o levantamento dos *frameworks* de recomendação foram os seguintes:

- Possuir código aberto;
- Ser implementado utilizado a linguagem Python em versão maior ou igual a Python3;
- Suportar o uso do algoritmo de recomendação considerando *datasets* customizados para geração da recomendação;
- Ser de fácil instalação no Sistema *Linux*.

A partir dos critérios descritos acima, foram selecionados os seguintes *frameworks*: Surprise, QREC, LightFM, Cornac, Case Recommender, LensKit, e Elliot. Estes *frameworks* são descritos nas seções que seguem.

3.1 Surprise

O Surprise (*Simple Python RecommendatIon System Engine*) (HUG, 2020), disponível em ⟨https://github.com/NicolasHug/Surprise⟩ é um kit de ferramentas para SciPy (Scikit) para implementação e análise de sistemas de recomendação que lidam com dados de avaliação explícita de itens. Até a redação deste documento, o Surprise integra 11 algoritmos de recomendação, além de diversas medidas de similaridade, como a similaridade por cosseno e a correlação de Pearson. Os algoritmos de predição suportados são: predição aleatória (usada para avaliação), quatro variantes da técnica KNN, algoritmos baseados em fatoração de matrizes (SVD, SVD++ e NMF) e os algoritmos baseados em FC SlopeOne e CoClustering.

O Surprise permite que a manipulação de *datasets* ocorra sem dificuldades, ao incluir *datasets* já bastantes conhecidos como o *Movielens*(HARPER; KONSTAN, 2015) e o *Jester*(GOLDBERG et al., 2001), ou permitindo que sejam utilizados *datasets* customizados pelos próprios usuários. O Suprise permite que estes *datasets* possam ser

representados em formato *.csv* ou utilizando os *dataframes* da biblioteca pandas para Python. Também é possível utilizá-lo através da linha de comando, sem a necessidade da implementação de um programa completo (HUG, 2020).

3.2 QREC

QRec(YU et al., 2021), disponível em ⟨https://github.com/Coder-Yu/QRec⟩ é um *framework* leve (*lightweight*) baseado na ferramenta *TensorFlow*, possuindo diversos algoritmos de recomendação implementados. São disponibilizadas as implementações de algoritmos de predição de avaliação de usuário e de itens (SlopeOne, PMF, SoRec, SVD++, RSTE, SVD, SocialMF, EE, SoReg, LOCALBAL, SREE, CUNE-MF, BPR, WRMF, SBPR, ExpoMF, CoFactor, TBPR, CDAE, DMF, NeuMF, CUNE-BPR, IR-GEN, SERec, APR, IF-BPR, CFGAN, NGCF, DiffNet, RSGAN, LightGCN, DHCF, ESRF, MHCN, SGL, SEPT, BUIR e SimGCL).

Algumas das características do *framework* são:

- **Multi-Plataforma**: o *framework* pode ser facilmente construído e utilizado em qualquer plataforma, incluindo *MS Windows*, *Linux* e *Mac OS*;
- **Rápida Execução**: *framework* baseado no *Numpy*, *TensorFlow* e outras estruturas *Lightweight*, ele possui uma performance rápida;
- **Configuração fácil**: a configuração dos modelos de recomendação são realizadas em arquivos de configuração próprios, permitindo a utilização de múltiplos protocolos de avaliação;
- **Fácil replicação**: através do conjunto de interfaces de recomendação implementadas pelo *framework*, novos algoritmos de recomendação podem ser facilmente implementados

Além de permitir que sejam utilizados diversos modelos de recomendação através de arquivos de configuração separados para a geração da recomendação, também pode ser utilizado através de *snippets* em Python para que a recomendação seja utilizada dentro de outras aplicações.

3.3 LightFM

O **LightFM**(KULA, 2015), disponível em ⟨https://github.com/lyst/lightfm⟩, implementa em Python diversos algoritmos populares de recomendação para uso com técnicas tanto de avaliação implícita quanto explicita, incluindo a implementação das funções BPR (*Bayesian personalized ranking*) e WARP (*Weighted Approximate-Rank Pairwise*).

De forma semelhante ao *Surprise*, este *framework* fornece *datasets* conhecidos como o *Movieslens* para que os algoritmos de recomendação sejam testados e validados. No entanto, a utilização de *datasets* que não são aqueles disponibilizados é possível. Para isto, é necessário que sejam realizados tratamentos dos dados de forma para que sejam reconhecidos pelo *framework*.

3.4 Cornac

O Cornac (SALAH; TRUONG; LAUW, 2020), disponível em ⟨https://github.com/PreferredAI/cornac⟩, é um *framework* para comparação de sistemas de recomendações multimodais, tendo como o foco tornar confortável trabalhar com modelos que possuem dados auxiliares que caracterização de itens, como descrição de texto, imagens, redes sociais, etc.

O *framework* foi um dos indicados pela ACM *Conference on Recommender Systems* 2023 para a avaliação e reprodução de sistemas de recomendação, permitindo experimentos rápidos e implementação direta de novos modelos. Além disso, este *framework* possui alta compatibilidade com bibliotecas de *machine learning* existentes, como o *TensorFlow* e *PyTorch*.

3.5 Case Recommender

O *framework* Case Recomender (COSTA et al., 2018), disponível em ⟨https://github.com/caserec/CaseRecommender⟩, é composto por diversas implementações em Python de algoritmos de recomendação populares, tanto para avaliação implícita quanto explicita. Entre os algoritmos existentes, 10 deles são utilizados para a recomendação de itens, sendo uma lista não exaustiva desses algoritmos: *BPRMF*, *ItemKNN*, *Item Attribute KNN*, *UserKNN*, etc. Além disso, possui 15 algoritmos para previsão de *ranking*, como Fatoração de Matrizes, *SVD*, *ItemKNN*, etc; e 2 algoritmos para *Clustering*, *Pacos* e *k-medois*.

O objetivo deste *framework* é fornecer um conjunto rico de componentes para auxiliar na construção de sistemas de recomendação customizados através dos diversos algoritmos citados, em conjunto com as técnicas de validação e avaliação embutidas.

É possível utilizar *datasets* próprios em conjunto com o mesmo, sendo de maneira semelhante ao *Surprise* e somente necessário se adequar ao padrão proposto pelo *framework*.

3.6 LensKit

O **LensKit** (EKSTRAND, 2020), disponível em ⟨https://lenskit.org/⟩, é um *toolkit* de código aberto antigamente implementado em Java e agora disponibilizado em Python que possui como foco a pesquisa e a educação em sistemas de recomendação. Ele disponibiliza suporte para o treinamento, a execução e a avaliação de algoritmos de recomendação.

Além dos *datasets* embutidos no *framework*, *MovieLens*, *ML100K*, *ML1M* e *ML10M*; o LensKit aceita outros *datasets* que incluam as informações necessárias para a recomendação.

3.7 Elliot

O **Elliot** (ANELLI et al., 2021), disponível em ⟨https://github.com/sisinflab/elliot⟩, é um *framework* de recomendação que analisa o problema da recomendação a partir da visão do pesquisador. A ideia é conduzir um experimento inteiro e por meio da alimentação do sistema com arquivos de configuração simples que auxiliem o *framework* na escolha da configuração. O **Elliot** destrincha a complexidade da combinação de estratégias de divisão, otimização do modelo de hiper-parâmetros, treinamento de modelos e geração de relatórios sobre os resultados do experimento.

A configuração do experimento de recomendação é toda feita em um arquivo separado, em que é definido quais dados serão utilizados, se os dados serão divididos, além de quais modelos e medidas de similaridade e avaliação deverão ser utilizados.

3.8 Escolha do *framework* de recomendação

Com os dados apresentados acima, é necessário escolher um desses *frameworks* para ser utilizado nesse trabalho. Após estudo comparativo dos *frameworks*, foi selecionado o *Surprise*. Visto que, embora os outros *frameworks* citados também apresentem funcionalidades necessárias para o trabalho, o *Surprise* se apresentou mais adequado para lidar com os *datasets* construídos para esta aplicação, além de que, comparado aos outros, apresenta uma documentação mais completa e estruturada para a consulta.

A tabela 3.8 presente abaixo apresenta uma comparação entre os *frameworks* de recomendação, a mesma foi utilizada para a escolha do *framework* mais adequado para esta implementação.

Framework	Foco do framework	Tipo de algoritmo	Medidas de similaridade
Surprise	Implementação de SR	Predição	X
QRec	Implementação de SR	Predição, Ranking de item	X
LightFM	Implementação de SR	Predição	X
Cornac	Comparação de SRs	Ranking de item	X
Case Recommender	Auxiliar na construção de SRs	Predição	X
LensKit	Avaliação de SR	Predição	X
Elliot	Conduzir experimentos com SR	Predição	X

4 Web Service de Recomendação: API e Arquitetura

Como apresentado no Capítulo 1, o objetivo deste projeto é o de projetar e implementar um sistema de recomendação baseado em uma arquitetura orientada a serviços na forma de um *Web Service* (WS) seguindo os padrões da API REST.

Este capítulo apresenta a arquitetura e API do *Web Service* de Recomendação proposto neste projeto. A primeira seção apresenta a API, em que são descritos seus métodos, funcionalidades e casos de uso. Em seguida, será apresentada a arquitetura do WS de recomendação, com a apresentação detalhada de cada um de seus módulos e suas relações com a API descrita na seção anterior.

4.1 Especificação da API do Web Service de recomendação

Na presente proposta, os dados trocados entre os clientes e o WS de recomendação são representados no formato JSON, o qual é uma solução independente de uma linguagem de programação/*framework* ou sistema operacional, facilitando a incorporação do cliente do serviço de recomendação com sistemas já existentes. O JSON foi adotado por ser considerado um formato mais simples e conciso do que o XML.

A especificação do WS proposto está presente no anexo B.

O WS proposto oferece métodos para a criação e manutenção de contextos e ações, além do registro e exclusão de itens e usuários, bem como a solicitação da recomendação. Antes da apresentação dos métodos propostos, este capítulo apresenta alguns conceitos básicos relacionados à API proposta.

4.1.1 Conceitos Básicos

Esta API considera uma série de definições de base para a requisição dos serviços, como contextos, ações do usuário, informações sobre os itens que o *Web Service* pode manter, métricas de avaliação (*ratings*) e algoritmos de recomendação oferecidos aos clientes do *Web Service*.

4.1.1.1 Contexto

Nesta proposta, cada sistema Web cliente do WS deve criar um contexto, o qual mantém todas as informações necessárias para a realização de recomendações aos usuários, considerando os itens oferecidos por este contexto. Por exemplo, se a Biblioteca

Digital de Literatura de Países Lusófonos (BLPL, ⟨https://www.literaturabrasileira.ufsc.br/⟩) utilizar a solução proposta, o administrado da BLPL deveria inicialmente criar um contexto no WS.

Cada contexto deve manter o seguinte:

- Identificador único;
- URL do sistema Web cliente;
- Informações do responsável;
- Dados de autenticação do cliente junto ao WS de recomendação;
- Associação a uma base de registros de todas as ações dos usuários, bem como informações sobre os itens a serem recomendados. Cada usuário e item do contexto devem ter um identificador único;
- Lista de algoritmos de recomendação escolhidos e seus respectivos parâmetros de configuração.

4.1.1.2 Autenticação

Conforme apresentado na Seção 2.2.1.2, existem algumas opções de métodos de autenticação do cliente no WS. Na presente proposta, o método adotado é a utilização de um *token de acesso*. Neste caso, será retornada uma chave de acesso no campo *X-Auth-Token* do cabeçalho após o contexto ter sido registrado no *Web Service*, sendo necessário que este *token* seja enviado no cabeçalho das requisições no campo *X-Auth-Token* para ser autenticado no sistema.

4.1.1.3 Registro de Ações

O WS proposto permite ao sistema Web cliente informar as interações dos usuários em um dado contexto, que é chamada aqui de ações. Cada tipo de ação é referenciada por um identificador único. Uma lista não exaustiva de ações incluem *Rating*, *Likes* e *Buy*.

Cada ação deve especificar o item que sofreu a mesma (avaliado, comprado e curtido) e uma lista opcional de parâmetros associados à ação (valor da avaliação). Para otimizar o montante de requisições ao WS, o sistema suporta o registro em conjunto de uma série de ações.

Cada registro de ações tem a seguinte estrutura:

- **ActionType**: identifica o tipo da ação que será registrada;
- **Unit**: identifica a unidade utilizada nessa ação;

- **itemSet**: contém os seguintes parâmetros responsáveis por relacionador os usuários com os itens:
  - **userId**: identifica o usuário que realizou a ação;
  - **itemId**: identifica o item sobre o qual a ação foi realizada;
  - **ParameterValue**: identifica o valor da ação registrada.

A estrutura JSON da Listagem B.2.1 ilustra a sintaxe do registro de avaliações de itens na BLPL.

```
ction": "Rating",
nit": "5-stars",
ontext_id": "BLPL",
temIds": [
{
  "item0001": [
    {
      "user": "user0001",
      "ParameterValue": 4
    },
    {
      "user": "user0002",
      "ParameterValue": 7
    },
    {
      "user": "user0003",
      "ParameterValue": 5
    }
  ],
  "item0002": [
    {
      "user": "user0002",
      "ParameterValue": 4
    },
    {
      "user": "user0004",
      "ParameterValue": 1
    }
  ]
}
```

4.1.1.4 Métricas

O WS permite ao cliente utilizar diferentes unidades já existentes no sistema. Estas unidades definem valores quantitativos relacionados à ação. Por exemplo, nas avaliações de itens, o contexto pode definir métricas como 5-estrelas, *likes*, etc. A estrutura JSON abaixo ilustra a criação da métrica 5-estrelas:

```
"Unit": "5-stars",
"maxScale": 1,
"minScale": 5
```

4.1.1.5 Registro de Itens

Cada ação já informa o identificador único dos itens (no contexto) que sofreram ações dos Sistemas Web clientes. Caso o cliente deseje utilizar algoritmos de recomendação que considerem características dos itens a recomendar, ele deve realizar o registro das características de todos os itens disponíveis no cliente e que sejam passíveis de recomendação.

A API proposta considera os seguintes tipos de itens e respectivas características:

- Item: tem como descrição mínima seu título (*title*) e descrição textual (*description*);
- Vídeo: além dos descritores gerais, um cliente do serviço pode descrever um vídeo utilizando os metadados atores (*actors*), diretores (*directors*), ano (*year*), tags (*tags*), gêneros (*genres*) e duração (*length*);
- Música: além dos descritores gerais, um cliente do serviço pode descrever uma música com os metadados cantores (*singers*), compositores (*composers*), ano (*year*), tags (*tags*), gêneros (*genres*) e duração (*length*);
- Livro: além dos descritores gerais, um cliente do serviço pode descrever um livro utilizando os descritores escritores (*writers*), ano (*year*), tags (*tags*), gêneros (*genres*) e número de páginas (*length*).

A estrutura JSON abaixo ilustra a descrição de um filme:

4.1.1.6 Algoritmos de Recomendação

O WS de recomendação proposto oferece a seus clientes um conjunto expansível de algoritmos de recomendação.

Exemplos de algoritmos incluem: recomendações baseadas em popularidade, previsão de *rating* (SVD, CoCluresting e KNNBasic) e baseada em conteúdo. A lista de algoritmos de recomendação disponíveis é publicada na documentação da API, sendo que cada algoritmo pode ser referenciado por um identificador único e possui uma série de parâmetros que podem ser customizados pelos clientes.

```
"id": "1274",
"title": "Akira",
"description": "In 2019, following a world war triggered by ...",
"directors": ["Katsuhiro Otomo"],
"actors": [
    "Mitsuo Iwata",
    "Nozomu Sasaki",
    "Mami Koyama"
],
"genres": [
    "Adventure",
    "Animation",
    "Sci-Fi",
    "Thriller"
]
```

Na criação do contexto, o cliente deverá informar a lista de algoritmos adotados com suas respectivas parametrizações. Por exemplo, para informar o uso da popularidade e o SVD, o cliente deve especificar a lista de algoritmos a seguir. No caso do algoritmo de popularidade, é necessário especificar as ações sobre os itens que contabilizam para a popularidade dos mesmos. Além disso, deve ser informado em minutos a periodicidade da atualização das listas de recomendação em cada algoritmo em (*updateTime*).

```
"id": "Popularity",
"updateTime": "00000", \\ 0 minutes
"parameters": {
  "actions": [
    "view",
    "buy"
  ]
}


"id": "SVD",
"updateTime": "00360", \\6 hours
"parameters": {
  "n_factors": 300,
  "n_epochs": 10,
  "biased": false
}
```

4.1.1.7 Base de Recomendações

Na presente arquitetura, a lista de recomendação para usuários e itens é mantida por uma base de recomendações. Esta base mantém uma lista para cada algoritmo de recomendação. Estas listas de recomendações são atualizadas periodicamente, sendo que esta periodicidade é definida pelo cliente no momento da criação do contexto.

A Listagem B.2.5 apresenta a representação da lista de recomendações para um dado usuário com o id *151*:

```
{
    "ranking":{
        "100k-ml":{
            "rating":{
                "SDV":
                    {
                        "151": [      // User Id
                            "661",
                            "914",
                            "3408",
                            "2355",
                            "1197",
                            "1287",
                            "2804",
                            "594",
                            "919",
                            "595"
                        ]
                    }
            }
        }

    }
}
```

A Listagem B.2.6 apresenta a representação da lista de recomendação para um dado item com id *1274*

```
{
    "item":{
        "100k-ml":{
            "content-based":{
                "genre":
                    {
                        "1274": [      // Item Id
                            "741",
                            "1025",
                            "1151",
                            "1205",
                            "2137",
                            "1287",
                            "783",
                            "334",
                            "664",
                            "1593"
                        ]
                    }
            }
        }

    }
}
```

4.1.1.8 Adaptação de Modelos

A arquitetura proposta considera uma representação de dados sobre usuários, itens e listas de recomendação, independentes da implementação do algoritmo de representação. Isto permite que diferentes implementações dos algoritmos de recomendação possam ser adaptadas à arquitetura proposta.

Esta operação de adaptação da representação de dados é realizada pelos chamados **Adaptadores de Modelos**, que devem ser implementados considerando a forma de representação do *dataset* usado pelo algoritmo de recomendação, também considerando o formato do resultado do algoritmo de recomendação.

Um exemplo disso é que um SR baseado na predição de *rankings* pode precisar que sejam passados como parâmetros quatro tipos de dados para que sejam realizadas a recomendação: id dos usuários, id dos itens, *rating* atribuídos ao itens e a escala utilizada na avaliação. Os dados que estão armazenados no WS não possuem a escala que é utilizada e outras informações que não serão utilizadas pelo SR, como o *timestamp* de quando os dados foram inseridos no WS. O adaptador de modelos serve para auxiliar que os SRs tenham todos os dados essenciais para que seja realizada a recomendação, sem dados desnecessários.

Outro exemplo da utilização do adaptador de modelos é o retorno de um SR.

Cada implementação possui um tipo de retorno diferente, mas o usuário do WS espera que quando ele peça uma recomendação, não importando o tipo de algoritmo que foi utilizado, ela venha em um formato padronizado. O adaptador de modelos é responsável por pegar as respostas do SRs e adaptá-las a um formato padrão para ser recebido pelo usuário.

Há duas funções que são utilizadas para que a representação dos dados seja adaptada.

A função *adapt_input* requere os dados ao atualizador de contextos e realiza o procedimento. Uma representação de ações adaptada será representada por três listas contendo o id de usuários, id de itens e os *ratings* dos itens, além de ser adicionada a representação da escala que será utilizada na recomendação por *ranking*.

Em uma representação de itens adaptada, são removidas as informações que não serão utilizadas para calcular as recomendações, como por exemplo o id dos itens. Após isso, os dados são filtrados e as palavras-vazias são removidas, tais como *"the"* e *"and"*, umas vez que são consideradas inúteis e não devem influenciar na recomendação. O último passo é preencher os dados em branco com um texto em branco.

O adaptador também é responsável por adaptar os resultados das recomendações através da função *adapt_output*, o que envolve transformar os itens retornados em uma lista numerada baseada em relevância, que contenha informações sobre o contexto, o tipo de recomendação e outros dados específicos de cada tipo de como recomendação, como o algoritmo utilizado para a recomendação baseada em *ranking* ou sobre qual categoria foi utilizada a recomendação baseada em itens.

4.1.2 Métodos de Criação e Remoção de Contexto

Os métodos que são responsáveis pelo gerenciamento dos contextos são os seguintes:

- **createContext**: permite a criação de um novo contexto de recomendação;
  - **URL de acesso:** /api/context
  - **Método:** POST
  - **Parâmetros do corpo:**
    - Context_id: Parâmetro que identifica o contexto que será criado;
    - url: URL do Sistema Web Cliente;
    - email: Email que será registrado junto do contexto;
    - name: Nome do responsável pela criação do contexto;
    - units: Métricas utilizadas no contexto, conforme descrito na Seção 4.1.1.4;

  - ∗ recset: Lista que identifica os algoritmos de recomendação e seus parâmetros, como apresentado na Seção 4.1.1.6.

- **Mensagem de resposta:**
  - ∗ 200 X-Auth-Token: TOKEN;
- **Mensagem de erro:**
  - ∗ 400 Bad Request;
  - ∗ 401 Unauthorized - Invalid user token;
  - ∗ 422 Unprocessable Entity - Context already exists;
  - ∗ 500 Internal Server Error.
- **Exemplo de Requisição:**

```
$ curl -X 'POST' \
    'http://127.0.0.1:8000/api/context' \
    -H 'accept: */*' \
    -H 'Content-Type: application/json' \
    -d '{
        "context_id": "BLPL",
        "url": "https://www.literaturabrasileira.ufsc.br",
        "email": "hans.buss@mailfence.com",
        "name": "Hans Buss",
        "recset": [
            {
                "id": "Popularity",
                "updateTime": "0s",
                "parameters": {
                    "actions": [
                        "view",
                        "buy"
                    ]
                }
            },
            {
                "id": "SVD",
                "updateTime": "6h",
                "parameters": {
                    "n_factors": 300,
                    "n_epochs": 10,
                    "biased": false
                }
            }
        ],
        "units": [
            {
                "Unit": "5-stars",
                "maxScale": 1,
                "minScale": 5
            }
        ]
    }'
```

– **Exemplo de Resposta positiva:**

```
$HTTP/1.1 200 OK
Content-Type: application/json
Date: Mon, 28 Dez 2023 14:09:34 GMT
Content-Length: 3
Connection: keep-alive'
x-auth-token: 7a14b8beb4cdcc2d01e3b68d828cd08c

{

    "X-Auth-Token": '7a14b8beb4cdcc2d01e3b68d828cd08c'
}
```

- **deleteContext**: permite a remoção de um contexto de recomendação existente. Quando um contexto é deletado, todas os seus dados, como ações e itens, são também removidos do WS de recomendação.
  - **URL de acesso:** /api/context/*Context_id*
  - **Método:** DELETE
  - **Parâmetros da url:**
    - Context_id: Parâmetro que identifica o contexto que será deletado;
  - **Mensagem de resposta:**
    - 200 Context deleted successfully;
  - **Mensagem de erro:**
    - 400 Bad Request;
    - 401 Unauthorized - Invalid user token;
    - 422 Unprocessable Entity - Context doesn't exists;
    - 500 Internal Server Error.
  - **Exemplo:**

```
$ curl -X 'DELETE' \
    'http://127.0.0.1:8000/api/context/BLPL' \
    -H 'accept: */*' \
    -H 'X-Auth: CONTEXTKEY'
```

4.1.3 Métodos para Registro de Ações

A API oferece os seguintes métodos para registro de ações no WS de recomendação:

- **registerAction**: permite o registro das ações que os usuários realizam no sistema Web cliente;
  - **URL de acesso:** /api/action/
  - **Método:** POST
  - **Parâmetros do corpo:**
    - Context_id: Parâmetro que identifica o contexto em que a ação será registrada;
    - actions: Conjunto de registros de ações, como definido na Seção 4.1.1.3;
  - **Mensagem de resposta:**
    - 200 Action registered successfully;
  - **Mensagem de erro:**
    - 400 Bad Request;
    - 401 Unauthorized - Invalid user token;
    - 422 Unprocessable Entity - Context dosen't exists;
    - 422 Unprocessable Entity - Action already exists;
    - 500 Internal Server Error.
  - **Exemplo:**

```
curl -X 'POST' \
    'http://127.0.0.1:8000/api/action/' \
    -H 'accept: */*' \
    -H 'X-Auth: CONTEXTKEY' \
    -H 'Content-Type: application/json' \
    -d '{
       "context_id": "BLPL",
       "actions": [
       {
            "Action": "Rating",
            "Unit": "5-stars",
            "Context_id": "BLPL",
            "itemIds": [
                {
                    "item0001": [
                    {
                        "user": "user0001",
                        "ParameterValue": 4
                    },
                    {
                        "user": "user0002",
                        "ParameterValue": 7
                    },
                    {
                        "user": "user0003",
                        "ParameterValue": 5
                    }
                    ],                                          "item0002": [
                    {
                        "user": "user0001",
                        "ParameterValue": 4
                    },
                    {
                        "user": "user0004",
                        "ParameterValue": 1
                    }
                ]
            }
        ]
    }
]
}
```

4.1.4 Métodos para Registro e Atualização de Itens

Os registros de itens a recomendar permitem ao sistema cliente informar características ou conteúdos de dados associados aos itens. Os métodos relacionados ao gerenciamento de itens são os seguintes:

- **registerItem** permite que sejam registrados itens no contexto;
  - **URL de acesso:** /api/item/*Context_id*/*itemId*
  - **Método:** POST

– **Parâmetros do body:**

* Context_id: Parâmetro que identifica o contexto em que o item será criado;
* itensSet: Parâmetro que contém o conjunto de itens sendo registrados, cuja estrutura está descrita na Seção 4.1.1.5.

– **Mensagem de resposta:**

* 200 Item registered successfully;

– **Mensagem de erro:**

* 400 Bad Request;
* 401 Unauthorized - Invalid user token;
* 422 Unprocessable Entity - Context doesn't exists;
* 422 Unprocessable Entity - Item schema doesn't exists;
* 422 Unprocessable Entity - Item already exists;
* 500 Internal Server Error.

– **Exemplo:**

```
            curl -X 'POST' \
                'http://127.0.0.1:8000/api/item/registerItem/' \
                -H 'accept: */*' \
                -H 'X-Auth: CONTEXTKEY' \
                -H 'Content-Type: application/json' \
                -H 'Content-Type: application/json' \
                -d '{
"context_id": "BLPL",
"itensSet": [
    {
        "id": "item0001",
        "title": "Akira",
        "description": "In 2019, following a world war triggered by ...",
        "directors": [
            "Katsuhiro Otomo"
        ],
        "actors": [
            "Mitsuo Iwata",
            "Nozomu Sasaki",
            "Mami Koyama"
        ],
        "genres": [
            "Adventure",
            "Animation",
            "Sci-Fi",
            "Thriller"
        ]
    }
]
```

– **updateItem** permite a atualização das características de itens no contexto;

* **URL de acesso:** /api/item/*Context_id*/*itemId*
* **Método:** POST
* **Parâmetros do body:**
  - Context_id: Parâmetro que identifica o contexto em que o item será criado;
  - itensSet: Parâmetro que contém o conjunto de itens sendo registrados, cuja estrutura está descrita na Seção 4.1.1.5.
* **Mensagem de resposta:**
  - 200 Item registered successfully;
* **Mensagem de erro:**
  - 400 Bad Request;
  - 401 Unauthorized - Invalid user token;
  - 422 Unprocessable Entity - Context doesn't exist;
  - 422 Unprocessable Entity - Item doesn't exist;
  - 422 Unprocessable Entity - Item schema doesn't exist ;
  - 500 Internal Server Error.
* **Exemplo:**

```
            curl -X 'POST' \
                'http://127.0.0.1:8000/api/item/updateItem/' \
                -H 'accept: */*' \
                -H 'X-Auth: CONTEXTKEY' \
                -H 'Content-Type: application/json' \
                -H 'Content-Type: application/json' \
                -d '{
"context_id": "BLPL",
"itensSet": [
    {
        "id": "item0001",
        "title": "Akira",
        "description": "In 2019, following a world war triggered by ...",
        "directors": [
            "Katsuhiro Otomo"
        ],
        "actors": [
            "Mitsuo Iwata",
            "Nozomu Sasaki",
            "Mami Koyama"
        ],
        "genres": [
            "Adventure",
            "Animation",
            "Sci-Fi",
            "Thriller"
        ]
    }
]
```

– **removeItem**, para a remoção de itens e todas as ações realizadas sobre o mesmo existentes no contexto;

  * **URL de acesso:** /api/item/*Context_id*/*userId*/*itemId*/
  * **Método:** DELETE
  * **Parâmetros da url:**
    · Context_id: Parâmetro que identifica o contexto em que está localizado o item a ser atualizado;
    · itemId: Id responsável por identificar o item que será deletado
  * **Mensagem de resposta:**
    · 200 Item removed successfully;
  * **Mensagem de erro:**
    · 400 Bad Request;
    · 401 Unauthorized - Invalid user token;
    · 422 Unprocessable Entity - Item dosen't exists;
    · 422 Unprocessable Entity - Context dosen't exists;
    · 500 Internal Server Error.

* **Exemplo:**

```
$ curl -X 'DELETE' \
    'http://127.0.0.1:8000/api/item/BLPL/item00001' \
    -H 'accept: */*' \
    -H 'X-Auth: CONTEXTKEY'
```

4.1.5 Remoção de Usuários

O método *removeUser* descrito a seguir permite remover todos os registros de ações de um determinado usuário.

– **removeUser** remove todas as ações de determinado usuário;

* **URL de acesso:** /api/item/*Context_id*/*userId*/
* **Método:** DELETE
* **Parâmetros da url:**
  - Context_id: Parâmetro que identifica o contexto que está localizado o item a ser atualizado;
  - userId: Id que identifica o usuário no qual serão apagadas as ações;
* **Mensagem de resposta:**
  - 200 History deleted successfully;
* **Mensagem de erro:**
  - 400 Bad Request;
  - 401 Unauthorized - Invalid user token;
  - 422 Unprocessable Entity - User dosen't exists;
  - 422 Unprocessable Entity - Context dosen't exists;
  - 500 Internal Server Error.
* **Exemplo:**

```
    'http://127.0.0.1:8000/api/item/BLPL/user0001/' \
    -H 'accept: */*' \
    -H 'X-Auth: CONTEXTKEY'
```

4.1.6 Método para solicitar recomendações

Os pedidos de recomendação podem ser solicitados através do método *getRecommendation* descrito a seguir. Este método retorna uma lista com as recomendações baseadas nos parâmetros enviados na requisição, que são: o tipo da recomendação, o id do item ou usuário alvo da recomendação e o número de recomendações desejada.

- **getRecommendation**;
  - **URL de acesso:** /api/recommendation/*Context_id*/*rectype*/*id*/*nresult*
  - **Método:** GET
  - **Parâmetros da url:**
    - Context_id: Parâmetro que identifica o contexto que está localizado o item a ser atualizado;
    - id: Id que identifica o item ou usuário alvo da recomendação. Aqui o servidor saberia se o ID é de item ou usuário pelo tipo da recomendação;
    - nresult: Quantidade de itens que serão retornados;
    - rectype: Tipo da recomendação, sendo um dos ids de algoritmos que se encontram presentes na documentação do Web *service*.
  - **Parâmetros da Query:**
- **Modelo de resposta:**

```
{
    "recommendationType": "SVD",
    "recommendationId": 1,
    "Recommendation": {

        "item0002": "0.987",
        "item0454": "0.982",
        "item0434": "0.972"
        }
}
```

- **Mensagem de erro:**
  - 400 Bad Request;
  - 401 Unauthorized - Invalid user token;
  - 422 Unprocessable Entity - Item doesn't exists;
  - 422 Unprocessable Entity - User doesn't exists;
  - 422 Unprocessable Entity - Context doesn't exists;
  - 422 Unprocessable Entity - Rectype doesn't exists;
  - 422 Unprocessable Entity - Ratingtype doesn't exists;
  - 500 Internal Server Error.
- **Exemplo:**

```
$ curl -X 'GET' \
    'http://127.0.0.1:8000/api/recommendation/BDLP/SVD/user0001/10
    /BLPL/userId
    /0/0' \
    -H 'accept: */*' \
    -H 'X-Auth: CONTEXTKEY'
```

- **Resposta do Método**: Uma lista, no formato JSON com os itens solicitados.

4.2 Arquitetura do WS de Recomendação

Esta seção apresenta a arquitetura de implementação do WS de Recomendação. A Figura 4.2.1 apresenta os principais componentes da arquitetura proposta.

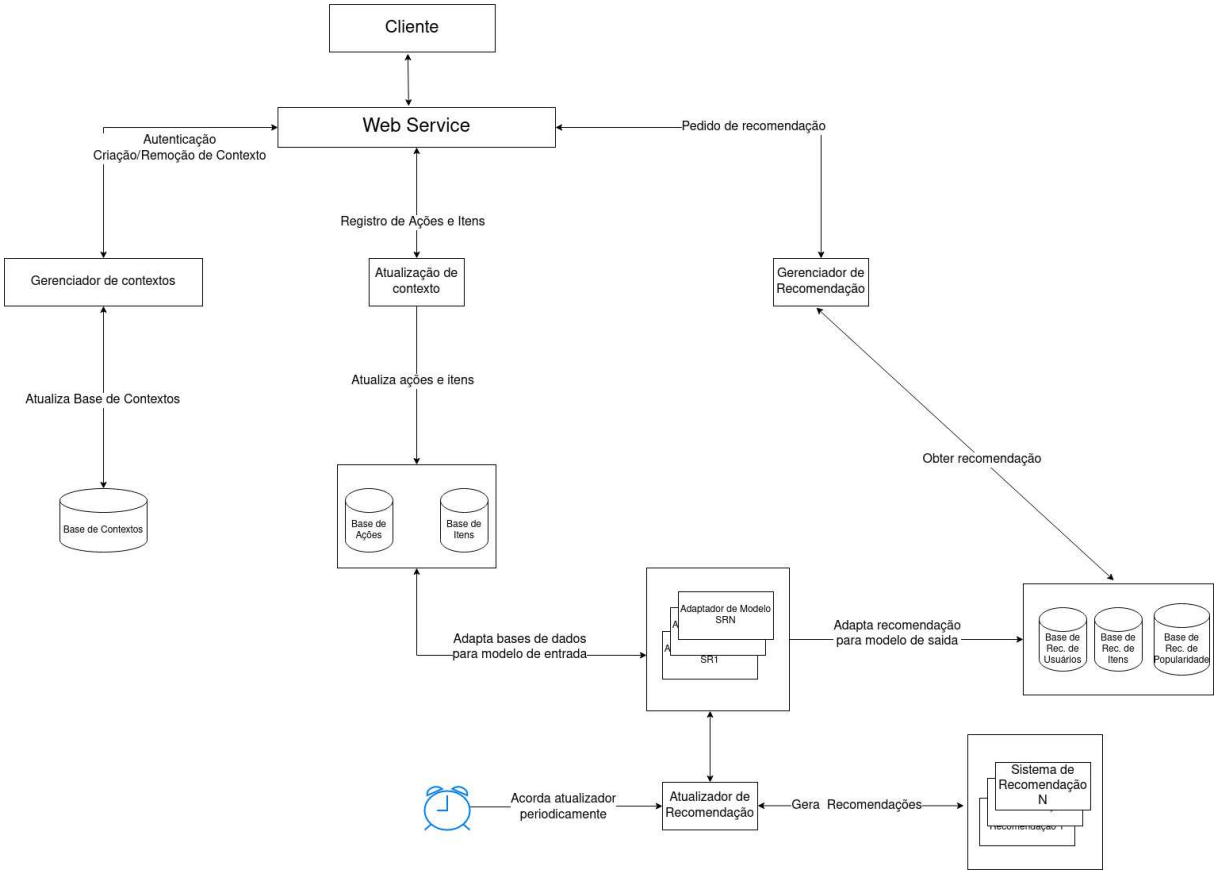

Figura 4.2.1 – Diagrama da arquitetura proposta

A arquitetura é composta pelos seguintes módulos que serão explicados com mais detalhes nas seções a seguir:

- Gerenciador de Contexto: responsável pela criação, configuração e remoção de contextos. Também é responsável pela autenticação do cliente;

- Atualizador de Contextos: armazena os dados das ações e itens utilizados para recomendação;
- Gerenciador de Recomendação: gerencia a entrega de recomendações para o cliente;
- Atualizador de Recomendações: responsável por gerar e armazenar novas recomendações periodicamente;
- Adaptador de Modelos: adapta os dados para a utilização nos SR e as saídas do SR para o cliente.

As subseções que seguem descrevem cada um destes componentes.

4.2.1 Componente Web Service

Este componente implementa todos os métodos apresentados na Seção 4.1, atuando como *endpoint* do WS de recomendação. Para tanto, ele utiliza serviços dos demais componentes da arquitetura.

4.2.2 Gerenciador de Contexto

O **Gerenciador de Contexto** é o modulo responsável pela criação e remoção de contextos, além de realizar a autenticação dos sistemas Web clientes. Ele implementa os serviços oferecidos pelos métodos da API *createContext* e *deleteContext* apresentados na Seção 4.1.2. Para persistência, este módulo utiliza a base de contextos, com informações sobre os contextos de recomendação criados.

A título de exemplo, a Figura 4.2.2 apresenta o diagrama de sequência para o método *createContext*, em que o sistema Web cliente realiza a requisição para o componente **Web Service**, que checa a sintaxe do pedido e em seguida encaminha a solicitação ao **Gerenciador de Contexto** para persistência e geração de chaves. Este último retorna o resultado do serviço ao componente **Web Service**, que finalmente responde ao cliente.

Conforme já apresentado, na criação do contexto são gerados *tokens* de autenticação utilizando os parâmetros usados para o registro do contexto, permitindo que o sistema Web cliente se autentique no WS de recomendação. Na arquitetura proposta, o módulo Gerenciador de Contexto também tem a função de autenticação dos clientes dos demais métodos oferecidos pela WS de recomendação com base no identificador do contexto e na chave de autenticação.

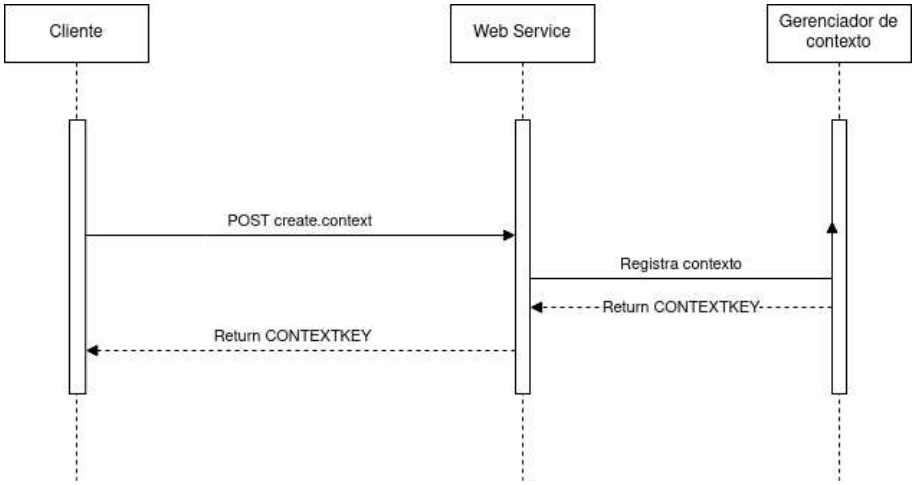

Figura 4.2.2 – Diagrama da sequência da criação de um contexto

4.2.3 Atualizador de Contextos

O **Atualizador de Contextos** é responsável por oferecer os serviços dos métodos para registro de ações (Seção 4.1.3) e de registro e atualização de itens (Seção 4.1.4). Para tal, ele realiza a persistência das ações e de características dos itens de todos os contextos de recomendação.

As funções da API relacionadas ao Atualizador de Contextos são:

- **registerAction**;
- **deleteAction**;
- **registerItem**;
- **updateItem**.

A figura 4.2.3 abaixo apresenta um diagrama de sequência com os passos necessários para o registro de ações e itens:

O primeiro passo para o registro de uma ação é o envio de uma requisição ao módulo **Web Service**, pedindo que seja registrado um grupo de ações ou itens. A fim de validar o cliente, o Web Service envia um pedido de autenticação do cliente para o **Gerenciador de Contexto**. Uma vez validado, o módulo **Web Service** encaminha o pedido de registro de ações ou itens para o atualizador de contexto. Este último é responsável pela checagem dos parâmetros e persistência destes dados.

4.2.4 Gerenciador de Recomendação

O **Gerenciador de Recomendação** é responsável por atender aos pedidos de recomendações dos sistemas Web Clientes, implementando o serviço oferecido pelo método

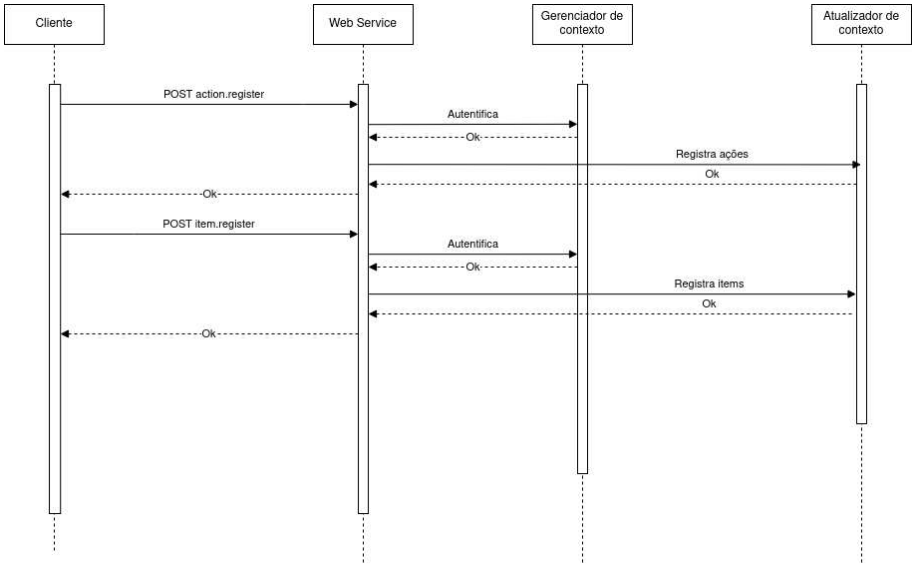

Figura 4.2.3 – Diagrama da sequência do registro de uma ação e um item

getRecommendation.

A arquitetura proposta utiliza uma solução de recomendação em que os algoritmos de recomendação são executados com certa periodicidade e as recomendações são persistidas em uma base de recomendação. A base de recomendação mantém, de forma estática, a lista de recomendações para todos os usuários e itens de todos os contextos. Este processo de atualização das recomendações será apresentado nas seções seguintes.

A Figura 4.2.4 apresenta, a título de exemplo, o processo de tratamento do método *getRecommendation* para um usuário identificado por *u0001*. Como visto neste diagrama de sequência, após receber a requisição, o módulo **Web Service** deve autenticar o cliente via o **Gerenciador de Contexto**. Uma vez autenticado, o **Web Service** solicita ao Gerenciador de Recomendação a lista de itens a recomendar ao usuário *user0001* e em seguida monta a resposta do pedido e encaminha ao cliente.

4.2.4.1 Atualizador de Recomendações

O módulo **Atualizador de Recomendações** é responsável pela atualização periódica das recomendações de usuários e itens mantidas na base de recomendações.

O período de atualização das listas de recomendação é dependente do contexto e são definidas no momento da criação do mesmo. Mais especificamente, cada algoritmo de recomendação pode ter um período de atualização diferente.

A Figura 4.2.5 apresenta o diagrama de sequência do processo de atualização da base de recomendações e considera a atualização da recomendação considerando um único algoritmo de recomendação, no exemplo, SVD. O processo de atualização compreende o seguinte:

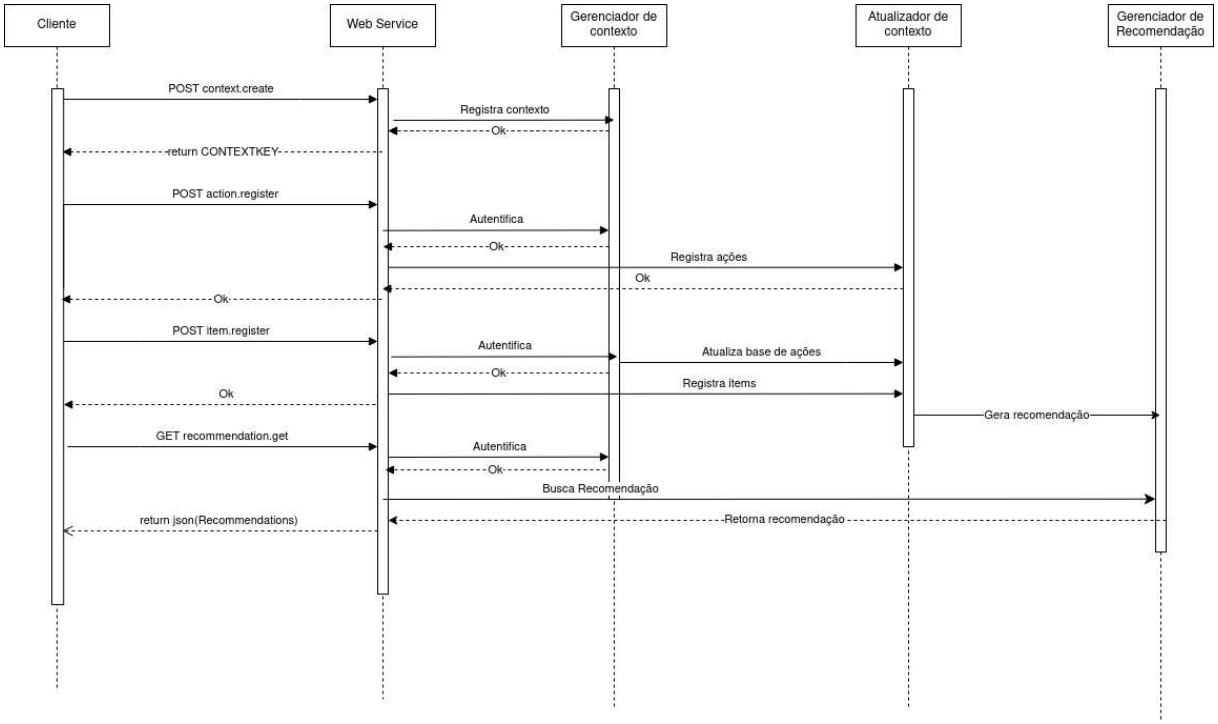

Figura 4.2.4 – Diagrama de Sequência do Web *Service*

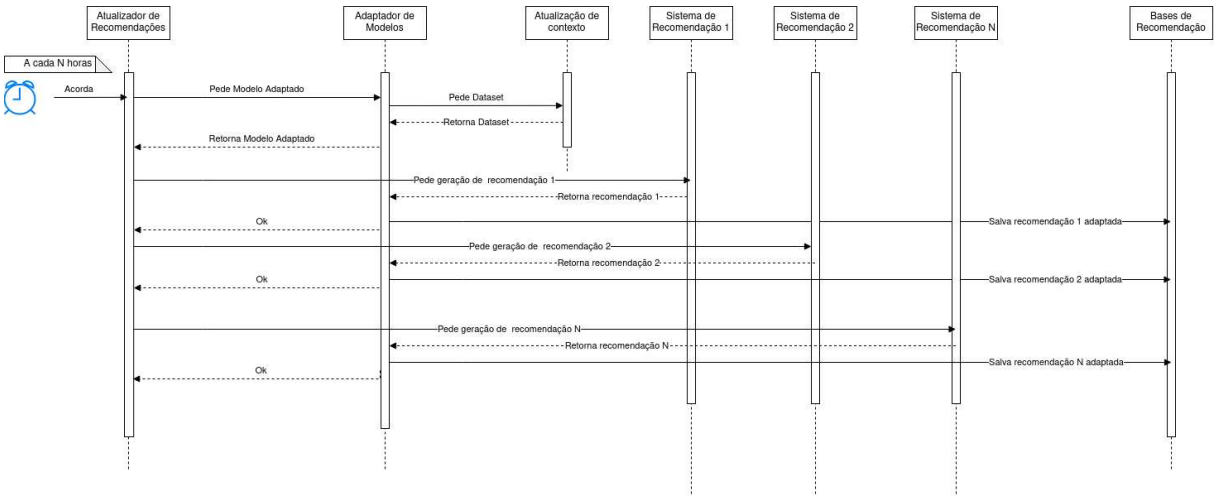

Figura 4.2.5 – Diagrama de Sequência da geração de uma recomendação

- O **Atualizador de Recomendações** é ativado periodicamente para atualizar a base de recomendação para o algoritmo;
- Ao ser ativado, o **Atualizador de Recomendações** deve gerar o *dataset* de entrada para o Sistema de Recomendação SVD utilizando os adaptador de modelo (Seção 4.1.1.8. Para tal, o **Atualizador de Recomendações** invoca o módulo **Adaptador de Modelo** do algoritmo SVD, responsável por adaptar os registros de ações (ou itens, dependendo do algoritmo de recomendação) ao formato de en-

trada implementado na recomendação SVD;

- Ao receber o *dataset* de entrada e considerando os parâmetros do algoritmo de recomendação especificados no contexto, o **Atualizador de Recomendações** lança o módulo **Sistema de Recomendação** SVD;
- Ao receber o resultado do Sistema de Recomendação, o **Atualizador de Recomendações** deve requisitar ao **Adaptador de Modelo** a adaptação do resultado do sistema de recomendação para o formato JSON considerado na arquitetura (descrito na Seção 4.1.1.7, devendo ser persistido na base de recomendações.

4.2.4.2 Adaptador de Modelos

Conforme descrito na Seção 4.1.1.8, um módulo **Adaptador de Modelo** é responsável pela conversão da representação interna de ações e itens para o formato esperado para um determinado Sistema de Recomendação. Existe um **Adaptador de Modelo** para cada algoritmo de recomendação.

Os módulos **Adaptadores de Modelo** também são responsáveis pela conversão do formato de saída do algoritmo de recomendação para o formato de persistência utilizada na base de recomendações (Seção 4.1.1.7.

.

4.2.4.3 Sistema de Recomendação

A arquitetura proposta foi desenvolvida de forma a suportar a uma lista expansível de sistemas de recomendação aos clientes. Cada módulo **Sistema de Recomendação** implementa um algoritmo de recomendação, que com base no dataset de entrada, gera a recomendação para usuários ou itens (dependendo do tipo de algoritmo).

Para ser integrada à arquitetura, uma solução de sistema de recomendação deve ser adaptada para prover dois métodos padrões:

- *generate_RecommendationXXXX*: Função chamada periodicamente para a geração da recomendação;
- *get_RecommendationXXXX*: Função chamada para o retorno da recomendação

Nesse exemplo *XXXX* é usado para sinalizar o campo onde deve estar presente o tipo da recomendação

4.3 Considerações Finais

Nesse capítulo foram apresentados os principais conceitos necessários para o entendimento e a utilização do Web Service proposto, além disso foi introduzida a arquitetura do sistema, onde foram explicados cada um de seus componentes, a sua utilização e métodos. No próximo capítulo é apresentada a implementação prova-de-conceito de um sistema que adota a arquitetura apresentada no presente capítulo.

5 Web *Service* de Recomendação: Prova de Conceito

Para avaliação da proposta do Web Service de Recomendação, cuja API e arquitetura foram descritas no Capítulo 5, é necessária uma implementação do sistema. Este capítulo apresenta a implementação prova-de-conceito do WS de Recomendação proposto.

A partir do protótipo desenvolvido, foram realizados alguns testes visando comprovar o seu correto funcionamento.

5.1 Escolha da tecnologia a ser utilizada

Para a seleção da tecnologia de desenvolvimento Web para a implementação do protótipo, foi realizado um estudo comparativo dos *frameworks* mais populares. Segue uma lista não exaustiva dos *frameworks* analisados:

- Django;
- Flask;
- Falcon;
- Connexion;
- Fastapi;
- Hug.

5.1.1 Django

O Django ⟨https://www.djangoproject.com/⟩ é um *framework* Web que segue um padrão MVC *Model-View-Controller* extremamente completo, que pode ser utilizado sozinho para a criação de APIs REST(Django Software Foundation,). Com o uso da extensão *Django REST framework* ⟨https://www.django-rest-framework.org/⟩ é possível integrar ao Django outras funcionalidades, como a autenticação de usuários, a limitação da quantidade de usuários online, e o mapeamento de bancos de dados relacionais para recursos da API.

Outras vantagens do *framework* Django incluem o fato do versionamento da API ser feita de uma forma flexível, ao permitir que múltiplos formatos de URL sejam utilizados com versões distintas. Além disso, o Django permite gerar páginas HTML para

a execução de todos os endpoints da API, permitindo que usuários e desenvolvedores executem pedidos diretamente de seus navegadores.

5.1.2 Flask

O micro framework Flask ⟨https://flask.palletsprojects.com/⟩ entrega uma versão inicial mais simples, com uma grande extensibilidade, onde as funções necessárias para determinada implementação devem ser adicionadas. Para isto, o desenvolvedor pode utilizar o rico ecossistemas de plugins disponíveis para o micro *framework* (GRINBERG, 2018).

Através do uso de plugins, é possível incorporar ao projeto funcionalidades presente no Django, como o uso de de autenticação dos usuários, a limitação do número de usuários conectados simultaneamente e o mapeamento de bancos de dados relacionais para a API, como por exemplo os modelos de banco de dados SQLAlchemy que o framework possui compatibilidade.

5.1.3 Connexion

O Connexion ⟨https://connexion.readthedocs.io/⟩ é um *framework* desenvolvido para operar sobre o Flask utilizando uma estratégia *API-first*, no qual a API é especificada antes utilizando uma linguagem de especificação padrão, sendo nesse caso utilizado o padrão OpenAPI/Swagger(SANTOS et al., 2023).

Após a especificação da API, o *framework* mapeia os endpoints declarados para as funções implementadas. Desta forma, não há necessidade de mapear funções aos endpoints e combinar a lógica da aplicação com a lógica REST em um mesmo arquivo.

Devido ao fato do *framework* ser construído sobre Flask, é possível que sejam utilizadas as mesmas funcionalidades, como por exemplo, utilizar plugins do ecosistema Flask para a adição de novos componentes ao código.

5.1.4 Falcon

O Falcon ⟨https://falconframework.org/⟩ é um lightweight micro *framework* com foco no desempenho e compatibilidade, graças à sua natureza minimalista e sua capacidade de ser combinado com ferramentas como *CPython* ou *Pypy* (GITHUB. . . , a).

O *framework* Falcon encoraja o uso do padrão estrutural REST ao empregar o uso de handler de métodos HTTPs para responder aos pedidos e permite que sejam utilizadas funções assíncronas nativamente ao oferecer suporte ao modelo *asyncio*.

5.1.5 FastAPI

O FastAPI ⟨https://fastapi.tiangolo.com/⟩ é um *framework* para o desenvolvimento de APIs, sendo considerado um dos mais rápidos do Python. Uma de suas vantagens é a possibilidade de resolver os pedidos de forma assíncrona (GITHUB..., c).

O *FastAPI* possuem diferenças significativas em relação ao textitFlask, como por exemplo o *FastAPI* não possui um servidor de desenvolvimento próprio, sendo necessário que sejam utilizados servidores como o *Daphne* ou *Uvicorn*. Outras diferenças incluem a não necessidade de uso da validação dos dados de forma explicita pelo desenvolvedor para garantir que os mesmos sejam do tipo correto.

5.1.6 Hug

O Hug ⟨https://www.hug.rest/⟩ é um micro *framework* para o desenvolvimento de APIs conhecido por sua alta performance, conseguindo responder a 3000 pedidos por segundo, ultrapassando *frameworks* como o *Flask*.

Grande parte do desempenho do Hug é devido ao fato dele ser compilado utilizando o *Cython*, um compilador que permite a utilização de código Python para a geração de código C com performance semelhante a mesma (GITHUB..., b).

Uma limitação do *framework* Hug é a falta de integração a um sistema de banco de dados, sendo necessária a utilização de bibliotecas externas. De toda forma, o Hug possibilita que a implementação seja abstraída do *framework* e utilizada em outros tipos de aplicações que não sejam necessariamente um Web Service.

5.1.7 Seleção do *framework*

Para o desenvolvimento do protótipo do WS de recomendação foi selecionado o *framework* Connexion. Apesar de existirem *frameworks* que oferecerem um melhor desempenho, o Connexion oferece outras vantagens que justifiquem essa escolha.

A possibilidade de estar utilizando uma estratégia API-First permite que o momento de implementação do sistema seja divido em duas partes. Primeiro será descrita a lógica utilizada pela API utilizando os padrões da *OpenAPI* e através do *framework* os recursos declarados serão mapeados para as funções implementadas na segunda parte, onde o foco será a implementação das técnicas da recomendação escolhidas sem a necessidade de estar mexendo na implementação da API para isto.

Outro aspecto considerado para a utilização do *framework* Connection é a possibilidade de serem utilizados diversos plugins disponíveis. Devido ao fato deste *framework* ter sido construído sobre o *framework* Flask, é possível estender suas funcionalidades

facilmente, dado ao grande número de opções existentes no ecossistema de plugins do mesmo.

Finalmente, a escolha do Connexion também é justificada pela possibilidade de adoção de uma estratégia de API-First em conjunto com as funcionalidades presentes em um dos *frameworks* mais utilizados.

5.2 Protótipo desenvolvido

O protótipo prova-de-conceito implementa algumas das funcionalidade do WS de Recomendação especificado no Capítulo 5.

5.2.1 API

No módulo **Web Service** foram implementados os métodos essenciais para o funcionamento da API listados abaixo:

- **createContext**;
- **deleteContext**;
- **registerAction**;
- **registerItem**;
- **getRecommendation**

5.2.2 Cliente Implementado

Para a avaliação do WS implementado, um protótipo de cliente foi desenvolvido para simular a utilização do WS. O protótipo foi desenvolvido para utilizar o dataset *ML-1m*, disponível dentro do *framework* Surprise.

O desenvolvimento protótipo de cliente envolveu duas etapas: A configuração do contexto de recomendação e o envio do dataset *ML-1m* para o WS.

Foi registrado um contexto chamado *"Ml-1m"* e configurado o SR de ranking para atualizar as recomendações utilizando o algoritmo SVD a cada 6h. Além disso foi definido que a métrica que será utilizada pelo contexto seria uma métrica de 5 estrelas, com as notas variando de 1 a 5.

Com o contexto tendo sido configurado, foi realizado o registro das ações presentes no dataset utilizado. O dataset é composto por quatro colunas contendo informações sobre o id do usuário, id do item, nota dada pelo usuário e o timestamp dessas ação. Os primeiros itens do dataset podem ser visualizado na tabela 5.2.2 a seguir.

userId	itemId	rating	timestamp
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116

O processo de registro de ações ao WS foi realizado através de uma leitura linha por linha do dataset, onde a cada linha era realizada uma requisição de registro de ação com o item do usuário, item do item e rating presentes.

5.2.3 Gerenciador de Contexto

Já no módulo **Gerenciador de Contexto** foram implementadas as principais funções relacionadas ao gerenciamento de contextos, como por exemplo o método *createContext* que permite que sejam criados novos contextos e gera um token de autenticação para ser utilizado nos demais métodos. A função *deleteContext* também foi implementada e permite que os contexos existentes possam ser deletados no Web Service.

A autenticação foi implementada utilizando os padrões propostos de utilização do header *X-Auth-Token* na requisição e é autenticado pela função *apikey_auth* do gerenciador. A autenticação embora seja funcional utiliza o header *X-Auth-Token* que é considerado um header não padronizado, sendo mais recomendado estar utilizando um header padronizado como o *Authorization*.

5.2.4 Sistemas de Recomendação implementados

Para essa pesquisa foram implementados cinco algoritmos de recomendação, sendo um deles baseado em popularidade, três deles baseados no rating dos itens e o último baseado no conteúdo dos itens.

O algoritmo baseado em popularidade foi implementado utilizando o número de ratings de cada item registrado no sistema, contando o número de ratings que cada item possível e aqueles que possuem o maior número de ratings são considerados os mais populares e são recomendados.

Os algoritmos que são considerados somente baseados em ratings foram implementados utilizando os algoritmos disponiveis no *framework Surprise*, foram eles: *SDV*, *CoClurestingo* e *KNNBasic*. Todos seguem a mesma estrutura, primeiro o conjunto de treinamento é gerado a partir da do dataset adaptado da base de ações e então cada algoritmo é treinado com o conjunto de treinamento. Após os algoritmos terem sido treinados, é gerado o conjunto de teste para que sejam obtidas as predições do mesmo. Com as predições geradas, as mesmas são mapeadas para cada usuário e retornadas as mais altas.

O algoritmo baseado em contéudo foi implementado sem a utilização de um *framework* focado em recomendação, no entanto foi utilizado a biblioteca de Machine Learning *Scikit-Learn*.

A implementação envolve utilizar um dos campos do item, como por exemplo o titulo. A primeira etapa do algoritmo é aplicar um algoritmo conhecido como Tf-idf (*Term frequency–inverse document frequency*) que verificar o grau de importância das palavras presentes nos campos do item, além de remover palavras que são consideradas inúteis, como por exemplo *"the"* e *"and"*. O resultado desse algoritmo é vector balanceado que representa o perfil dos itens e que será utilizado em conjunto com a similaridade do cosseno.

Com a frequência de cada palavra tendo sido calculada, é gerada a matriz utilizada no calculo da similaridade do cosseno e calculado a similaridade. Após isso o algoritmo itinera por todos os itens do dataset e recomenda os itens que possuem o menor ângulo do cosseno, ignorando aqueles que possuem um ângulo muito aberto.

5.2.5 Adaptadores de Modelo

Para realizar esta avaliação, foi necessário que os resultados obtidos através do Web Service fossem comparados com os obtidos pelo *framework* de recomendação Surprise, executado de maneira independente, dentro de uma maquina local, sem a utilização das funções implementadas no Web service.

Para que a avaliação fosse realizada através do Web Service, foi necessário implementar o adaptador de modelos propostos no capítulo anterior.

Para os dados de entrada, a implementação do adaptador envolveu remover colunas que não são utilizadas pelo SR, como por exemplo a de *timestamps* que era adicionada assim que uma ação era registrada no sistema. Além disso, o Surprise necessita que o dataframe possuam uma escala para realizar a recomendação, então o adaptador faz o trabalho de inserir a escala definida no momento da criação do contexto no dataframe.

A adaptação dos resultados para que sejam enviados ao cliente foi implementada pegando as recomendações salvas pelo SR e pegando as primeiras N recomendações, sendo N o número de recomendações que o cliente solicitou receber. O adaptador então gera um json contendo o tipo do algoritmo utilizado, o contexto que solicitou a recomendações e armazena por ordem de relevância as recomendações solicitadas.

5.3 Avaliação do Web Service Implementado

Após a implementação ter sido finalizada, uma rodada de testes foi realizada visando avaliar o correto funcionamento da proposta. Foram realizados dois modelos

de testes que são abordados a seguir: Testes de funcionalidade dos métodos e teste de funcionalidade da recomendação.

5.3.1 Avaliação da funcionalidade dos métodos

Os testes de funcionalidade dos métodos implementados foram realizados de forma empírica e buscavam analisar se métodos implementados funcionavam de maneira correta. A avaliação de medidas de desempenho como tempo de resposta ou escalabilidade não foram o objetivo desse projeto e ocorrerão em trabalhos futuros, com a utilização de máquinas mais adequadas do que a utilizada nesse projeto.

A realização dos testes ocorreu conforme os métodos eram implementados. Primeiro foram realizados requisições para a criação, configuração e a exclusão de contextos. Após isso foram implementados os métodos para registros de ações e de itens, ambos os métodos foram testados e apresentaram o comportamento esperado.

O método de pedido de recomendação foi analisado e seus testes são apresentados com maiores detalhes na sessão a seguir.

5.3.2 Avaliação da funcionalidade da recomendação

Para permitir a realização dos testes da funcionalidade da recomendação, foi desenvolvido um Sistema Web cliente simples, no domínio de filmes, e considerando os dados disponíveis no dataset *MovieLens-1m* disponível no *framework* Surprise. Para tal, este dataset foi adaptados à representação JSON adotada pelo WS de Recomendação.

Após a criação do contexto de recomendação para o domínio de filmes, o cliente implementado simula o registro de parte das ações do dataset *MovieLens-1m* usando o método de registro de ações do WS de Recomendação.

Com as ações registradas no WS de Recomendação, o adaptador de modelos foi utilizado para adaptar as ações em um dataset utilizado para a recomendação SDV.

Para a comparação dos testes realizados no cliente desenvolvido, o dataset *MovieLens-1m* foi carregado utilizando o *framework* Surprise e seguidos mesmos passos a partir daqui.

A realização dos testes ocorreu utilizando o sobre o dataset uma técnica de validação cruzada, um procedimento utilizado para avaliar modelos de machine learning com uma amostra da dados limitada. A técnica de validação cruzada escolhida é conhecida como *K-Fold* e envolve em embaralhar o dataset de forma aleatória e então separar o dataset em k grupos, sendo possível a utilização dados diferentes para os conjuntos de testes e de treino.

Após a divisão do dataset em cinco grupos, foi iterado por cada uma dessas cinco subdivisões realizando os procedimentos realizados pelo algoritmo de recomendação SDV,

como o treino dos conjuntos de teste e treino e a geração das predições de recomendação. Após as predições terem sido geradas, elas foram utilizadas para calcular a *precisão* e o *recall*.

A *precisão* e o *recall* são métricas de desempenho baseadas na relevância em que auxiliam no processo de avaliar a corretude do WS proposto. A *precisão* pode ser definida como a métrica que indica dentro do total de itens recomendados, quantos realmente eram relevantes. Já o *recall* é a métrica que indica quantos dos itens considerados relevantes foram recomendados.

As tabelas 1 e 2 apresentam as métricas de precisão e recall respectivamente calculadas utilizando o Surprise e o Web Service.

Tabela 1 – Dados de Precisão

Iterações	Precision	
	Surprise	Web Service
1	0.7753742060204464	0.7786408302970199
2	0.7800530152418929	0.7750469405787603
3	0.7801403159871969	0.7786282306163134
4	0.77469473451573	0.7815333222903301
5	0.7747514910536885	0.7778449561040367

Tabela 2 – Dados de Recall

Iterações	Recall	
	Surprise	Web Service
1	0.27462589325184045	0.27603624905146623
2	0.2763249924721947	0.276546566030835
3	0.27708402378390456	0.27454630268195096
4	0.2758626608206604	0.27496932779808037
5	0.2736571650639147	0.2775289624587902

Analisando as medidas obtidas, é possível perceber que os resultados obtidos pelo Surprise e pelo Web Service são semelhantes e comprova que o WS implementado apresenta o comportamento esperado em relação a geração de recomendações.

A taxa de precisão obtida sinaliza que a implementação consegue estar realizando a recomendação de um número altos de itens considerados relevantes para o usuário. No entanto, a taxa de recall mostra que embora vários itens relevantes são recomendados, somente um terço dos itens considerados relevantes são realmente recomendados.

6 Conclusão

O desenvolvimento de uma API REST pra Sistemas de Recomendação possibilitou que fosse realizado uma revisão do estado da arte de Sistemas de Recomendação e Serviços Web, aliado ao levantamento de implementações em código aberto de Web Services e Sistemas de Recomendação. Esse levantamento permitiu que fossem selecionados as tecnologias e algoritmos mais adequados para esse desenvolvimento.

A avaliação do WS desenvolvido possibilitou verificar o funcionamento correto de cada um dos métodos implementados, além a avaliação das recomendações permitiu comparar o funcionamento do WS implementado com o do framework utilizado e verificar a corretude na geração de recomendações.

6.1 Trabalhos futuros

Recomenda-se que sejam estudadas formas de realizar a desacoplação dos algoritmos de recomendação do Web Service de recomendação através de WS dedicados para cada algoritmo, permitindo que as operações de alto custo computacional não sejam realizadas no servidor principal, e sim em servidores dedicados não causando o sobrecarregamento do mesmo.

Em conjunto a desacoplação dos algoritmos de recomendação que já se encontram implementados, é possível e recomendado que sejam adicionados novos algoritmos de recomendação ao Web Service, seja adaptando algoritmos existentes no Surprise ou a utilização de outros frameworks.

Além disso, é necessários realizar testes de avaliação de desempenho do WS implementado, como por exemplo métricas de tempo de resposta e escalabidade, em que máquinas adequadas para esses testes do que a utilizada nesse projeto.

Referências

AGGARWAL, C. C. Recommender systems. 2016.

AÏMEUR, E. et al. Wsrs: A web service recommender system. In: *International Conference on Web Information Systems and Technologies*. [s.n.], 2007. Disponível em: ⟨https://api.semanticscholar.org/CorpusID:5987914⟩.

ANELLI, V. W. et al. Elliot: A comprehensive and rigorous framework for reproducible recommender systems evaluation. In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, 2021. (SIGIR '21), p. 2405–2414. ISBN 9781450380379. Disponível em: ⟨https://doi.org/10.1145/3404835.3463245⟩.

BAELDUNG, W. b. *Intro to RAML - the restful API modeling language*. 2022. Disponível em: ⟨https://www.baeldung.com/raml-restful-api-modeling-language-tutorial⟩.

BOBADILLA, J. et al. Recommender systems survey. 2013.

BOUTEMEDJET, S.; ZIOU, D. A graphical model for context-aware visual content recommendation. *Multimedia, IEEE Transactions on*, v. 10, p. 52 – 62, 02 2008.

BURKE, R. Hybrid web recommender systems. In: *The Adaptive Web*. [S.l.: s.n.], 2007.

CASAGRANDE, M. F. R. *TÉCNICA DE RECOMENDAÇÃO PARA REPOSITÓRIOS DIGITAIS BASEADA EM METADADOS E AGRUPAMENTO DE USUÁRIOS*. Tese (Doutorado) — Centro Tecnológico, Universidade Federal de Santa Catarina, 2014. Disponível em: ⟨https://repositorio.ufsc.br/handle/123456789/181250⟩. Acesso em: 2023-06-07.

CAZELLA, S. C.; NUNES, M. A. S. N.; REATEGUI, E. B. A ciência da opinião: Estado da arte em sistemas de recomendação. 2010.

CHEN, M.; LAPAUGH, A. S.; SINGH, J. P. Predicting category accesses for a user in a structured information space. In: *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, 2002. (SIGIR '02), p. 65–72. ISBN 1581135610. Disponível em: ⟨https://doi.org/10.1145/564376.564390⟩.

CHERNEV, A.; BöCKENHOLT, U.; GOODMAN, J. Choice overload: A conceptual review and meta-analysis. 2015.

CLAYPOOL, M. et al. Combining content-based and collaborative filters in an online newspaper. In: *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. [S.l.: s.n.], 1999.

COSTA, A. da et al. Case recommender: A flexible and extensible python framework for recommender systems. In: *Proceedings of the 12th ACM Conference on Recommender Systems*. New York, NY, USA: ACM, 2018. (RecSys '18), p. 494–495. ISBN 978-1-4503-5901-6. Disponível em: ⟨http://doi.acm.org/10.1145/3240323.3241611⟩.

COSTA, E.; AGUIAR JANDERSON E MAGALHãES, J. Sistemas de recomendação de recursos educacionais: conceitos, técnicas e aplicações. 2013.

Django Software Foundation. *Django*. Disponível em: ⟨https://djangoproject.com⟩.

EKSTRAND, M. D. Lenskit for python: Next-generation software for recommender systems experiments. In: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. New York, NY, USA: Association for Computing Machinery, 2020. (CIKM '20), p. 2999–3006. ISBN 9781450368599. Disponível em: ⟨https://doi.org/10.1145/3340531.3412778⟩.

FERRIS, C.; FARRELL, J. What are web services? *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 46, n. 6, p. 31, jun 2003. ISSN 0001-0782. Disponível em: ⟨https://doi.org/10.1145/777313.777335⟩.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000. Disponível em: ⟨https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm⟩. Acesso em: 2023-06-10.

FURTADO, C. A. et al. Custom ordering on digital library information retrieval. In: *Proceedings of the XV Brazilian Symposium on Multimedia and the Web*. New York, NY, USA: Association for Computing Machinery, 2009. (WebMedia '09). ISBN 9781605588803. Disponível em: ⟨https://doi.org/10.1145/1858477.1858505⟩.

GAO, C. et al. A survey of graph neural networks for recommender systems: Challenges, methods, and directions. *ACM Trans. Recomm. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 1, n. 1, mar 2023. Disponível em: ⟨https://doi.org/10.1145/3568022⟩.

GITHUB - falconry/falcon: The no-magic web data plane API and microservices framework for Python developers, with a focus on reliability, correctness, and performance at scale. — github.com. ⟨https://github.com/falconry/falcon⟩. [Accessed 07-10-2023].

GITHUB - hugapi/hug: Embrace the APIs of the future. Hug aims to make developing APIs as simple as possible, but no simpler. — github.com. ⟨https://github.com/hugapi/hug⟩. [Accessed 07-10-2023].

GITHUB - tiangolo/fastapi: FastAPI framework, high performance, easy to learn, fast to code, ready for production — github.com. ⟨https://github.com/tiangolo/fastapi⟩. [Accessed 07-10-2023].

GOLDBERG, K. et al. Eigentaste: A constant time collaborative filtering algorithm. *Inf. Retr.*, Kluwer Academic Publishers, USA, v. 4, n. 2, p. 133–151, jul 2001. ISSN 1386-4564. Disponível em: ⟨https://doi.org/10.1023/A:1011419012209⟩.

GRINBERG, M. *Flask web development: developing web applications with python*. [S.l.]: "O'Reilly Media, Inc.", 2018.

HARPER, F. M.; KONSTAN, J. A. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, ACM New York, NY, USA, v. 5, n. 4, p. 1–19, 2015.

HENRIQUE, b. et al. *Documentação de Api no modelo de arquitetura rest*. 2022. Disponível em: ⟨https://dev.delivery/documentacao-api-rest/⟩.

HUG, N. Surprise: A python library for recommender systems. *Journal of Open Source Software*, The Open Journal, v. 5, n. 52, p. 2174, 2020. Disponível em: ⟨https://doi.org/10.21105/joss.02174⟩.

KAVANAGH, P. 2018. Disponível em: ⟨https://www.linkedin.com/pulse/three-disadvantages-tightly-coupled-connections-pavinee-kavanagh⟩.

KOREN, Y.; BELL, R.; VOLINSKY, C. Matrix factorization techniques for recommender systems. *Computer*, v. 42, n. 8, p. 30–37, 2009.

KULA, M. Metadata embeddings for user and item cold-start recommendations. In: BOGERS, T.; KOOLEN, M. (Ed.). *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.* CEUR-WS.org, 2015. (CEUR Workshop Proceedings, v. 1448), p. 14–21. Disponível em: ⟨http://ceur-ws.org/Vol-1448/paper4.pdf⟩.

LTD, T. P. *Introduction to API blueprint*. 2021. Disponível em: ⟨https://testfully.io/blog/api-blueprint/#what-is-api-blueprint⟩.

LUO, X. et al. An efficient non-negative matrix-factorization-based approach to collaborative filtering for recommender systems. *IEEE Transactions on Industrial Informatics*, v. 10, n. 2, p. 1273–1284, 2014.

MOONEY, R. J.; ROY, L. Content-based book recommending using learning for text categorization. In: *Proceedings of the Fifth ACM Conference on Digital Libraries*. New York, NY, USA: Association for Computing Machinery, 2000. (DL '00), p. 195–204. ISBN 158113231X. Disponível em: ⟨https://doi.org/10.1145/336597.336662⟩.

O que é uma API RESTful? Disponível em: ⟨https://aws.amazon.com/pt/what-is/restful-api/#seo-faq-pairs#what-are-restful-api-auth⟩.

PANDHARBALE, P.; MOHANTY, S. N.; JAGADEV, A. K. Study of recent web service recommendation methods. In: *2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*. [S.l.: s.n.], 2020. p. 692–695.

PARK, D. et al. A literature review and classification of recommender systems research. *Expert Systems with Applications*, Elsevier Ltd., v. 39, n. 11, p. 10059–10072, set. 2012. ISSN 0957-4174. Funding Information: This research is supported by Ministry of Culture, Sports and Tourism(MCST) and Korea Creative Content Agency(KOCCA) in the Culture Technology(CT) Research Developement Program 2011.

RESNICK, P.; VARIAN, H. R. Recommender systems. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 40, n. 3, p. 56–58, mar 1997. ISSN 0001-0782. Disponível em: ⟨https://doi.org/10.1145/245108.245121⟩.

RICCI, F.; ROKACH, L.; SHAPIRA, B. Recommender systems handbook. In: _____. [S.l.: s.n.], 2010. v. 1-35, p. 1–35. ISBN 978-0-387-85819-7.

RUOTSALO, T. Methods and applications for ontology-based recommender systems. 01 2010.

SALAH, A.; TRUONG, Q.-T.; LAUW, H. W. Cornac: A comparative framework for multimodal recommender systems. *Journal of Machine Learning Research*, v. 21, n. 95, p. 1–5, 2020.

SALAKHUTDINOV, R.; MNIH, A. Probabilistic matrix factorization. In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2007. (NIPS'07), p. 1257–1264. ISBN 9781605603520.

SALLES, A. Serviço web de recomendação baseado em ontologias e grafos para repositórios digitais. 2017.

SANTOS, J. et al. *spec-first/connexion*. 2023. Disponível em: ⟨https://github.com/spec-first/connexion⟩.

SARWAR, B. et al. Application of dimensionality reduction in recommender system – a case study. 08 2000.

TIRYAKI, A. M.; YüCEBAş, S. C. An ontology based product recommendation system for next generation e-retail. *Journal of Organizational Computing and Electronic Commerce*, Taylor Francis, v. 33, n. 1-2, p. 1–21, 2023. Disponível em: ⟨https://doi.org/10.1080/10919392.2023.2226542⟩.

WILLRICH, R. et al. Adaptive information retrieval system applied to digital libraries. In: . [S.l.: s.n.], 2006. v. 192, p. 165–173.

Wordnik. *Swagger: A simple, open standard for describing REST APIs with JSON*. 2012. Disponível em: ⟨http://swagger.wordnik.com/⟩.

YU, J. et al. Socially-aware self-supervised tri-training for recommendation. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery Data Mining*. [S.l.: s.n.], 2021. p. 2084–2092.

A Apêndice 1

Projeto e Implementação de uma API REST para Sistemas de Recomendação

Hans B. Heidemann[1], **Roberto Willrich**[2]

[1]Departamento de Informática e Estatística - Universidade Federal de Santa Catarina (UFSC)
Santa Catarina – SC – Brazil

hans.buss@grad.ufsc.br, roberto.willrich@ufsc.br

Resumo. *Atualmente, devido ao montante de informação, produtos e serviços disponibilizados em rede, é fundamental o oferecimento de serviços que realizam certos tipos de filtragem de informações de acordo com as preferências dos usuários. Esta é a função dos Sistemas de Recomendação (SRs), que são capazes de estimar os perfis dos usuários sobre um dado domínio e encaminhar recomendações personalizadas de itens que sejam potencialmente úteis para estes usuários. Muitos sistemas Web são adaptados para uso de um serviço de recomendação específico, preparando os dados de entrada do algoritmo de recomendação e adaptando o resultado para apresentação ao usuário final. Uma alternativa possivelmente mais interessante é o uso de serviços Web de recomendação, facilitando alterações/atualização do serviço de recomendação, sem grandes impactos no sistema Web. Neste sentido, o presente trabalho especifica uma API (Application Programming Interface) e desenvolve uma arquitetura de implementação de um serviço Web de recomendação em conformidade com o padrão da API-REST (API Representational State Transfer). Através destas API, qualquer sistema Web autenticado pode se registar, atualizar o perfil dos usuários e solicitar recomendações para um dado usuário. Para avaliar a proposta, foi desenvolvido um protótipo prova-de-conceito da API de recomendação, que foi testada a partir de um serviço Web cliente simulado no contexto de recomendações de filmes e utilizando o dataset Movielens. O protótipo implementa três métodos de recomendação, sendo eles por popularidade, baseado em predição de avaliação dos itens pelo usuário, e baseado em conteúdo dos itens. A partir dos testes realizados, foi possível observar o funcionamento adequado do Serviço Web de recomendação proposto neste trabalho.*

1. Introdução

Cada vez mais os usuários da Web têm a sua disposição uma enorme quantidade de recursos digitais, além de serviços e produtos. Devido ao montante de itens disponíveis para os usuários da Web, a sobrecarga de informação é um problema recorrente, quando a tarefa do usuário em localizar itens relevantes se torna complicada, devido ao grande número de itens recuperados nas buscas na Web [Salles 2017]. Sistemas de Recomendação (SR) podem ser utilizados para solucionar este problema de sobrecarga de itens [Chernev et al. 2015], tendo como principal objetivo auxiliar no processo de identificação de itens relevantes aos interesses e preferências dos usuários [Cazella et al. 2010]. Tais sistemas auxiliam o usuário no processo de localização de

itens que sejam relevantes com base de técnicas de filtragem de informação, removendo informações que sejam irrelevantes ou desnecessárias à determinada pessoa. Na maior parte dos sistemas Web que se beneficiam de SR, a implementação dos serviços de recomendação ocorre de forma integrada (ou fortemente acoplada) aos demais módulos do sistema e ao seu banco de dados. À medida que a complexidade dessa implementação cresce, suas limitações começam a aparecer, como por exemplo, a alta dependência entre componentes do sistema, o alto custo necessário para a manutenção, além da pouca flexibilidade e reusabilidade, em que até mesmo uma pequena mudança pode afetar diversos componentes do sistema, limitando a adaptação do mesmo para uso em outros contextos [Kavanagh 2018]. O objetivo desse estudo foi a implementação de um Web Service de recomendação.

2. Fundamentação Teórica

2.1. Web Services

Como definido pelo W3C *Web Services Architecture Working Group*, um Serviço Web (WS - *Web service*) é uma aplicação de software identificada por uma URI, nas quais as interfaces são capazes de serem definidas, descritas e descobertas como artefatos XML [Ferris and Farrell 2003]. Tais aplicações permitem a interação direta com outros sistemas Web utilizando mensagens baseadas em XML e/ou JSON, trocadas via protocolos de Internet, normalmente utilizando HTTP/HTTPS, mas não restrito a tais, podendo serem utilizados outros mecanismo de transporte, como o FTP. As vantagens na utilização de WS incluem a desacoplação de interface do serviço da implementação, a vinculação dinâmica de serviços, além do aumento da interoperabilidade multilinguagem e multi-plataforma.

2.2. Representational State Transfer (REST)

Para [Fielding 2000], o REST é um estilo arquitetural que define 6 restrições relacionadas a estilos arquiteturais para sistemas de hipermídia distribuídos criado para guiar o projeto e o desenvolvimento da arquitetura da Web. Esse estilo ignora os detalhes dos componentes da implementação e da sintaxe do protocolo ao propor um conjunto de restrições de como a arquitetura de sistemas de hipermídia distribuídos, como a Web, devem funcionar.

2.3. Sistemas de Recomendação

Sistemas de recomendação (SR) estão sendo cada vez mais utilizados na Web, seja no momento da busca por um filme ou um item em um site de *e-commerce*, ou mesmo na hora de receber um *post* de um conhecido em uma rede social. SRs estão presentes nesses momentos com o objetivo apoiar a localização de itens que sejam do interesse do usuário em meio à quantidade gigantesca de itens que são disponíveis na Web. Desta forma, os SRs auxiliam em minimizar o problema chamado de *Sobrecarga de itens* [Salles 2017]. Existem várias definições para SRs, que variam dependendo do autor que se consulta. Para [Resnick and Varian 1997], um SR tem por objetivo apoiar o processo social de indicar ou receber recomendações, no qual se busca a identificação de um conjunto de itens com base nos interesses do usuário. Outros autores, como [Ruotsalo 2010], definem SR como um tipo especifico de sistemas de filtragem de informação, utilizados para a identificação de um conjunto de itens relevantes ao usuário.

3. Revisão Literaria

3.1. Frameworks de recomendação

Como colocado em [Salles 2017], *frameworks* são soluções que facilitam o reuso de bibliotecas na implementação de sistemas da recomendação, sendo necessária a configuração e adaptação dos dados dos itens e usuários para o *framework* em questão. Esse capítulo busca apresentar alguns dos principais *frameworks* existentes. Para isto, foi realizado um levantamento de *frameworks* citados em artigos científicos, além de pesquisar os disponíveis no site **Github**. Os critérios escolhidos para o levantamento dos *frameworks* de recomendação foram os seguintes:

- Possuir código aberto;
- Ser implementado utilizado a linguagem Python em versão maior ou igual a Python3;
- Suportar o uso do algoritmo de recomendação considerando *datasets* customizados para geração da recomendação;
- Ser de fácil instalação no Sistema *Linux*.

A partir dos critérios descritos acima, foram selecionados os seguintes *frameworks*: Surprise, QREC, LightFM, Cornac, Case Recommender, LensKit, e Elliot. Estes *frameworks* são descritos na tabela **??** abaixo.

Framework	Foco do framework	Tipo de algoritmo	Medidas de similaridade
Surprise	Implementação de SR	Predição	X
QRec	Implementação de SR	Predição, Ranking de item	X
LightFM	Implementação de SR	Predição	X
Cornac	Comparação de SRs	Ranking de item	X
Case Recommender	Auxiliar na construção de SRs	Predição	X
LensKit	Avaliação de SR	Predição	X
Elliot	Conduzir experimentos com SR	Predição	X

3.2. Web Framework

Para a seleção da tecnologia de desenvolvimento Web para a implementação do protótipo, foi realizado um estudo comparativo dos *frameworks* mais populares. Segue uma lista não exaustiva dos *frameworks* analisados:

- Django;
- Flask;
- Falcon;
- Connexion;
- Fastapi;
- Hug.

Para o desenvolvimento do protótipo do WS de recomendação foi selecionado o *framework* Connexion. Apesar de existirem *frameworks* que oferecerem um melhor desempenho, o Connexion oferece outras vantagens que justifiquem essa escolha. A possibilidade de estar utilizando uma estratégia API-First permite que o momento de implementação do sistema seja divido em duas partes. Primeiro será descrita a lógica utilizada pela API utilizando os padrões da *OpenAPI* e através do *framework* os recursos declarados serão mapeados para as funções implementadas na segunda parte, onde o foco será a implementação das técnicas da recomendação escolhidas sem a necessidade

de estar mexendo na implementação da API para isto. Outro aspecto considerado para a utilização do *framework* Connection é a possibilidade de serem utilizados diversos plug-ins disponíveis. Devido ao fato deste *framework* ter sido construído sobre o *framework* Flask, é possível estender suas funcionalidades facilmente, dado ao grande número de opções existentes no ecossistema de plugins do mesmo. Finalmente, a escolha do Connexion também é justificada pela possibilidade de adoção de uma estratégia de API-First em conjunto com as funcionalidades presentes em um dos *frameworks* mais utilizados.

4. Web Service de Recomendação: API e Arquitetura

Este capítulo apresenta a arquitetura e API do *Web Service* de Recomendação proposto neste projeto. A primeira seção apresenta a API, em que são descritos seus métodos, funcionalidades e casos de uso. Em seguida, será apresentada a arquitetura do WS de recomendação, com a apresentação detalhada de cada um de seus módulos e suas relações com a API descrita na seção anterior.

Esta API considera uma série de definições de base para a requisição dos serviços, como contextos, ações do usuário, informações sobre os itens que o *Web Service* pode manter, métricas de avaliação (*ratings*) e algoritmos de recomendação oferecidos aos clientes do *Web Service*.

Cada sistema Web cliente do WS deve criar um contexto, o qual mantém todas as informações necessárias para a realização de recomendações aos usuários, considerando os itens oferecidos por este contexto. Por exemplo, se a Biblioteca Digital de Literatura de Países Lusófonos (BLPL, `https://www.literaturabrasileira.ufsc.br/`) utilizar a solução proposta, o administrado da BLPL deveria inicialmente criar um contexto no WS.

Existem algumas opções de métodos de autenticação do cliente no WS. Na presente proposta, o método adotado é a utilização de um *token de acesso*. Neste caso, será retornada uma chave de acesso no campo *X-Auth-Token* do cabeçalho após o contexto ter sido registrado no *Web Service*, sendo necessário que este *token* seja enviado no cabeçalho das requisições no campo *X-Auth-Token* para ser autenticado no sistema.

O WS proposto permite ao sistema Web cliente informar as interações dos usuários em um dado contexto, que é chamada aqui de ações. Cada tipo de ação é referenciada por um identificador único. Uma lista não exaustiva de ações incluem *Rating*, *Likes* e *Buy*. Cada ação deve especificar o item que sofreu a mesma (avaliado, comprado e curtido) e uma lista opcional de parâmetros associados à ação (valor da avaliação). Para otimizar o montante de requisições ao WS, o sistema suporta o registro em conjunto de uma série de ações.

O WS permite ao cliente utilizar diferentes unidades já existentes no sistema. Estas unidades definem valores quantitativos relacionados à ação. Por exemplo, nas avaliações de itens, o contexto pode definir métricas como 5-estrelas, *likes*, etc.

Cada ação já informa o identificador único dos itens (no contexto) que sofreram ações dos Sistemas Web clientes. Caso o cliente deseje utilizar algoritmos de recomendação que considerem características dos itens a recomendar, ele deve realizar o registro das características de todos os itens disponíveis no cliente e que sejam passíveis de recomendação.

O WS de recomendação proposto oferece a seus clientes um conjunto expansível de algoritmos de recomendação. Exemplos de algoritmos incluem: recomendações baseadas em popularidade, previsão de *rating* (SVD, CoCluresting e KNNBasic) e baseada em conteúdo. A lista de algoritmos de recomendação disponíveis é publicada na documentação da API, sendo que cada algoritmo pode ser referenciado por um identificador único e possui uma série de parâmetros que podem ser customizados pelos clientes. Na criação do contexto, o cliente deverá informar a lista de algoritmos adotados com suas respectivas parametrizações. Por exemplo, para informar o uso da popularidade e o SVD, o cliente deve especificar a lista de algoritmos a seguir. No caso do algoritmo de popularidade, é necessário especificar as ações sobre os itens que contabilizam para a popularidade dos mesmos. Além disso, deve ser informado em minutos a periodicidade da atualização das listas de recomendação em cada algoritmo em (*updateTime*).

A arquitetura proposta considera uma representação de dados sobre usuários, itens e listas de recomendação, independentes da implementação do algoritmo de representação. Isto permite que diferentes implementações dos algoritmos de recomendação possam ser adaptadas à arquitetura proposta. Esta operação de adaptação da representação de dados é realizada pelos chamados **Adaptadores de Modelos**, que devem ser implementados considerando a forma de representação do *dataset* usado pelo algoritmo de recomendação, também considerando o formato do resultado do algoritmo de recomendação. Outro exemplo da utilização do adaptador de modelos é o retorno de um SR. Cada implementação possui um tipo de retorno diferente, mas o usuário do WS espera que quando ele peça uma recomendação, não importando o tipo de algoritmo que foi utilizado, ela venha em um formato padronizado. O adaptador de modelos é responsável por pegar as respostas do SRs e adaptá-las a um formato padrão para ser recebido pelo usuário.

5. Arquitetura do WS de Recomendação

Esta seção apresenta a arquitetura de implementação do WS de Recomendação. As subseções que seguem descrevem cada um destes componentes.

O **Gerenciador de Contexto** é o modulo responsável pela criação e remoção de contextos, além de realizar a autenticação dos sistemas Web clientes. Ele implementa os serviços oferecidos pelos métodos da API *createContext* e *deleteContext*. Para persistência, este módulo utiliza a base de contextos, com informações sobre os contextos de recomendação criados.

O **Atualizador de Contextos** é responsável por oferecer os serviços dos métodos para registro de ações e de registro e atualização de itens. Para tal, ele realiza a persistência das ações e de características dos itens de todos os contextos de recomendação.

O **Gerenciador de Recomendação** é responsável por atender aos pedidos de recomendações dos sistemas Web Clientes, implementando o serviço oferecido pelo método *getRecommendation*. A arquitetura proposta utiliza uma solução de recomendação em que os algoritmos de recomendação são executados com certa periodicidade e as recomendações são persistidas em uma base de recomendação. A base de recomendação mantém, de forma estática, a lista de recomendações para todos os usuários e itens de todos os contextos.

O módulo **Atualizador de Recomendações** é responsável pela atualização

periódica das recomendações de usuários e itens mantidas na base de recomendações. O período de atualização das listas de recomendação é dependente do contexto e são definidas no momento da criação do mesmo. Mais especificamente, cada algoritmo de recomendação pode ter um período de atualização diferente.

A arquitetura foi desenvolvida de forma a suportar a uma lista expansível de sistemas de recomendação aos clientes. Cada módulo **Sistema de Recomendação** implementa um algoritmo de recomendação, que com base no dataset de entrada, gera a recomendação para usuários ou itens (dependendo do tipo de algoritmo).

6. Protótipo desenvolvido

Para a avaliação do WS implementado, um protótipo de cliente foi desenvolvido para simular a utilização do WS. O protótipo foi desenvolvido para utilizar o dataset *ML-1m*, disponível dentro do *framework* Surprise. O desenvolvimento protótipo de cliente envolveu duas etapas: A configuração do contexto de recomendação e o envio do dataset *ML-1m* para o WS. Foi registrado um contexto chamado *"Ml-1m"* e configurado o SR de ranking para atualizar as recomendações utilizando o algoritmo SVD a cada 6h. Além disso foi definido que a métrica que será utilizada pelo contexto seria uma métrica de 5 estrelas, com as notas variando de 1 a 5. Com o contexto tendo sido configurado, foi realizado o registro das ações presentes no dataset utilizado.

O módulo **Gerenciador de Contexto** foram implementadas as principais funções relacionadas ao gerenciamento de contextos, como por exemplo o método *createContext* que permite que sejam criados novos contextos e gera um token de autenticação para ser utilizado nos demais métodos. A função *deleteContext* também foi implementada e permite que os contexos existentes possam ser deletados no Web Service. A autenticação foi implementada utilizando os padrões propostos de utilização do header *X-Auth-Token* na requisição e é autenticado pela função *apikey_auth* do gerenciador. A autenticação embora seja funcional utiliza o header *X-Auth-Token* que é considerado um header não padronizado, sendo mais recomendado estar utilizando um header padronizado como o *Authorization*.

Para essa pesquisa foram implementados cinco algoritmos de recomendação, sendo um deles baseado em popularidade, três deles baseados no rating dos itens e o último baseado no conteúdo dos itens. O algoritmo baseado em popularidade foi implementado utilizando o número de ratings de cada item registrado no sistema, contando o número de ratings que cada item possível e aqueles que possuem o maior número de ratings são considerados os mais populares e são recomendados. Os algoritmos que são considerados somente baseados em ratings foram implementados utilizando os algoritmos disponiveis no *framework Surprise*, foram eles: *SDV*, *CoCluresting* e *KNNBasic*. Todos seguem a mesma estrutura, primeiro o conjunto de treinamento é gerado a partir da do dataset adaptado da base de ações e então cada algoritmo é treinado com o conjunto de treinamento. Após os algoritmos terem sido treinados, é gerado o conjunto de teste para que sejam obtidas as predições do mesmo. Com as predições geradas, as mesmas são mapeadas para cada usuário e retornadas as mais altas. O algoritmo baseado em contéudo foi implementado sem a utilização de um *framework* focado em recomendação, no entanto foi utilizado a biblioteca de Machine Learning *Scikit-Learn*. A implementação envolve utilizar um dos campos do item, como por exemplo o titulo. A primeira etapa do

algoritmo é aplicar um algoritmo conhecido como Tf-idf (*Term frequency–inverse document frequency*) que verificar o grau de importância das palavras presentes nos campos do item, além de remover palavras que são consideradas inúteis, como por exemplo *"the"* e *"and"*. O resultado desse algoritmo é vector balanceado que representa o perfil dos itens e que será utilizado em conjunto com a similaridade do cosseno. Com a frequência de cada palavra tendo sido calculada, é gerada a matriz utilizada no calculo da similaridade do cosseno e calculado a similaridade. Após isso o algoritmo itinera por todos os itens do dataset e recomenda os itens que possuem o menor ângulo do cosseno, ignorando aqueles que possuem um ângulo muito aberto.

Para que a avaliação fosse realizada através do Web Service, foi necessário implementar o adaptador de modelos propostos no capítulo anterior. Para os dados de entrada, a implementação do adaptador envolveu remover colunas que não são utilizadas pelo SR, como por exemplo a de *timestamps* que era adicionada assim que uma ação era registrada no sistema. Além disso, o Surprise necessita que o dataframe possuam uma escala para realizar a recomendação, então o adaptador faz o trabalho de inserir a escala definida no momento da criação do contexto no dataframe. A adaptação dos resultados para que sejam enviados ao cliente foi implementada pegando as recomendações salvas pelo SR e pegando as primeiras N recomendações, sendo N o número de recomendações que o cliente solicitou receber. O adaptador então gera um json contendo o tipo do algoritmo utilizado, o contexto que solicitou a recomendações e armazena por ordem de relevância as recomendações solicitadas.

Após a implementação ter sido finalizada, uma rodada de testes foi realizada visando avaliar o correto funcionamento da proposta. Foram realizados dois modelos de testes que são abordados a seguir: Testes de funcionalidade dos métodos e teste de funcionalidade da recomendação.

Para permitir a realização dos testes da funcionalidade da recomendação, foi desenvolvido um Sistema Web cliente simples, no domínio de filmes, e considerando os dados disponíveis no dataset *MovieLens-1m* disponível no *framework* Surprise. Para tal, este dataset foi adaptados à representação JSON adotada pelo WS de Recomendação. Após a criação do contexto de recomendação para o domínio de filmes, o cliente implementado simula o registro de parte das ações do dataset *MovieLens-1m* usando o método de registro de ações do WS de Recomendação. Com as ações registradas no WS de Recomendação, o adaptador de modelos foi utilizado para adaptar as ações em um dataset utilizado para a recomendação SDV. Para a comparação dos testes realizados no cliente desenvolvido, o dataset *MovieLens-1m* foi carregado utilizando o *framework* Surprise e seguidos mesmos passos a partir daqui. A realização dos testes ocorreu utilizando o sobre o dataset uma técnica de validação cruzada, um procedimento utilizado para avaliar modelos de machine learning com uma amostra da dados limitada. A técnica de validação cruzada escolhida é conhecida como *K-Fold* e envolve em embaralhar o dataset de forma aleatória e então separar o dataset em *k* grupos, sendo possível a utilização dados diferentes para os conjuntos de testes e de treino. Após a divisão do dataset em cinco grupos, foi iterado por cada uma dessas cinco subdivisões realizando os procedimentos realizados pelo algoritmo de recomendação SDV, como o treino dos conjuntos de teste e treino e a geração das predições de recomendação. Após as predições terem sido geradas, elas foram utilizadas para calcular a *precisão* e o *recall*. A *precisão* e o *recall* são métricas de desempenho

baseadas na relevância em que auxiliam no processo de avaliar a corretude do WS proposto. A *precisão* pode ser definida como a métrica que indica dentro do total de itens recomendados, quantos realmente eram relevantes. Já o *recall* é a métrica que indica quantos dos itens considerados relevantes foram recomendados. As tabelas 1 e 2 apresentam as métricas de precisão e recall respectivamente calculadas utilizando o Surprise e o Web Service.

Table 1. Dados de Precisão

Iterações	Precision	
	Surprise	Web Service
1	0.7753742060204464	0.7786408302970199
2	0.7800530152418929	0.7750469405787603
3	0.7801403159871969	0.7786282306163134
4	0.77469473451573	0.7815333222903301
5	0.7747514910536885	0.7778449561040367

Table 2. Dados de Recall

Iterações	Recall	
	Surprise	Web Service
1	0.27462589325184045	0.27603624905146623
2	0.2763249924721947	0.276546566030835
3	0.27708402378390456	0.27454630268195096
4	0.2758626608206604	0.27496932779808037
5	0.2736571650639147	0.2775289624587902

Analisando as medidas obtidas, é possível perceber que os resultados obtidos pelo Surprise e pelo Web Service são semelhantes e comprova que o WS implementado apresenta o comportamento esperado em relação a geração de recomendações. A taxa de precisão obtida sinaliza que a implementação consegue estar realizando a recomendação de um número altos de itens considerados relevantes para o usuário. No entanto, a taxa de recall mostra que embora vários itens relevantes são recomendados, somente um terço dos itens considerados relevantes são realmente recomendados.

7. Conclusão

O desenvolvimento de uma API REST pra Sistemas de Recomendação possibilitou que fosse realizado uma revisão do estado da arte de Sistemas de Recomendação e Serviços Web, aliado ao levantamento de implementações em código aberto de Web Services e Sistemas de Recomendação. Esse levantamento permitiu que fossem selecionados as tecnologias e algoritmos mais adequados para esse desenvolvimento. A avaliação do WS desenvolvido possibilitou verificar o funcionamento correto de cada um dos métodos implementados, além a avaliação das recomendações permitiu comparar o funcionamento do WS implementado com o do framework utilizado e verificar a corretude na geração de recomendações.

8. Trabalhos futuros

Recomenda-se que sejam estudadas formas de realizar a desacoplação dos algoritmos de recomendação do Web Service de recomendação através de WS dedicados para cada algoritmo, permitindo que as operações de alto custo computacional não sejam realizadas no servidor principal, e sim em servidores dedicados não causando o sobrecarregamento do mesmo. Em conjunto a desacoplação dos algoritmos de recomendação que já se encontram implementados, é possível e recomendado que sejam adicionados novos algoritmos de recomendação ao Web Service, seja adaptando algoritmos existentes no Surprise ou a utilização de outros frameworks. Além disso, é necessários realizar testes de avaliação de desempenho do WS implementado, como por exemplo métricas de tempo de resposta e escalabidade, em que máquinas adequadas para esses testes do que a utilizada nesse projeto.

References

Cazella, S. C., Nunes, M. A. S. N., and Reategui, E. B. (2010). A ciência da opinião: Estado da arte em sistemas de recomendação.

Chernev, A., Böckenholt, U., and Goodman, J. (2015). Choice overload: A conceptual review and meta-analysis.

Ferris, C. and Farrell, J. (2003). What are web services? *Commun. ACM*, 46(6):31.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Kavanagh, P. (2018).

Resnick, P. and Varian, H. R. (1997). Recommender systems. *Commun. ACM*, 40(3):56–58.

Ruotsalo, T. (2010). Methods and applications for ontology-based recommender systems.

Salles, A. (2017). Serviço web de recomendação baseado em ontologias e grafos para repositórios digitais.

B Apêndice 2

B.1 Código Fonte

Os códigos a seguir estão disponíveis no seguinte repositorio do github ⟨https://github.com/hansb21/recommender_api⟩

B.1.1 Especificação da API proposta

```
1  agger.yml
2
3  api: 3.0.0
4  :
5  tle: "RP Flask REST API"
6  scription: "An API about people and notes"
7  rsion: "1.0.0"
8
9  ers:
10 url: "/api"
11
12 onents:
13 rameters:
14 Context:
15   name: "Context"
16   description: "Context used for recommendation"
17   in: path
18   required: True
19   schema:
20     type: "string"
21 Action:
22   name: "Action"
23   description: "Action used for recommendation"
24   in: path
25   required: True
26   schema:
27     type: "string"
28 itemId:
29   name: "itemId"
30   description: "Id of a item used on the recommendation"
31   in: path
32   required: True
33   schema:
34     type: "string"
```

```
35  userId:
36    name: "userId"
37    description: "Id of a user focus of the recommendation"
38    in: path
39    required: True
40    schema:
41      type: "string"
42  date:
43    name: "date"
44    description: "date of creation"
45    in: path
46    required: True
47    schema:
48      type: "string"
49  actionValue:
50    name: "actionValue"
51    description: "value of an action"
52    in: path
53    required: True
54    schema:
55      type: "integer"
56  minScale:
57    name: "minScale"
58    description: "Minimum value of rating scale"
59    in: path
60    required: True
61    schema:
62      type: "integer"
63  maxScale:
64    name: "maxScale"
65    description: "Maximum value of rating scale"
66    in: path
67    required: True
68    schema:
69      type: "integer"
70  nresult:
71    name: "nresult"
72    description: "number of items that shall be recommended"
73    in: path
74    required: True
75    schema:
76      type: "integer"
77  rectype:
78    name: "rectype"
79    description: "type of algorithm of recommendation that will be used"
80    in: path
81    required: True
```

```
    schema:
      type: "integer"
ratingtype:
    name: "ratingtype"
    description: "type of algorithm of recommendation that will be used in raking type
    ↪  recommendation"
    in: query
    required: False
    schema:
      type: "integer"
category:
    name: "category"
    description: "category that will be used in content-based type recommendation"
    in: query
    required: False
    schema:
      type: "string"

sponses:
NotFound:
    description: "The specified resource was not found"
Unauthorized:
    description: "Unauthorized"
InternalServerError:
    description: "Internal Server Error"
UnprocessableEntity:
    description: "Unprocessable Entity"
BadRequest:
    description: "Bad Request"

curitySchemes:
api_key:
    type: apiKey
    name: X-Auth
    in: header
    x-apikeyInfoFunc: app.apikey_auth

s:
ontext:
get:
    operationId: "context.read_all"
    tags:
      - "Context"
    summary: "Read the list of context"
    responses:
      "200":
        description: "Successfully read context list"
```

```
128       "400":
129         $ref: "#/components/responses/BadRequest"
130       "401":
131         $ref: "#/components/responses/Unauthorized"
132       "422":
133         $ref: "#/components/responses/UnprocessableEntity"
134       "500":
135         $ref: "#/components/responses/InternalServerError"
136     security:
137       - api_key: []
138
139   post:
140     operationId: "context.create"
141     tags:
142       - "Context"
143     summary: "Creates a new contex"
144     requestBody:
145       x-body-name: Context
146       required: true
147       content:
148         application/json:
149           schema:
150             type: object
151             required:
152               - Context
153             properties:
154               Context:
155                 type: string
156               name:
157                 type: string
158               email:
159                 type: string
160                 format: email
161               recommenders:
162                 type: array
163                 items:
164                   type: object
165                   properties:
166                     recommender:
167                       type: object
168                       properties:
169                         id:
170                           type: integer
171                           format: int32
172                         updateTime:
173                           type: integer
174                           format: int32
```

```
175             item_schema:
176               type: string
177   responses:
178     "200":
179       description: "Successfully createad context"
180     "400":
181       $ref: "#/components/responses/BadRequest"
182     "401":
183       $ref: "#/components/responses/Unauthorized"
184     "422":
185       $ref: "#/components/responses/UnprocessableEntity"
186     "500":
187       $ref: "#/components/responses/InternalServerError"
188   security:
189     - api_key: []
190
191 ontext/{Context}:
192 delete:
193   operationId: "context.delete"
194   tags:
195     - "Context"
196   summary: "Deletes an existing context"
197   parameters:
198     - $ref: "#/components/parameters/Context"
199   responses:
200     "200":
201       description: "Successfully deleted context"
202     "400":
203       $ref: "#/components/responses/BadRequest"
204     "401":
205       $ref: "#/components/responses/Unauthorized"
206     "422":
207       $ref: "#/components/responses/UnprocessableEntity"
208     "500":
209       $ref: "#/components/responses/InternalServerError"
210   security:
211     - api_key: []
212 etric/:
213 post:
214   operationId: "context.createScale"
215   tags:
216     - "Context"
217   summary: "Create a new scale that the user may use for recommendation"
218   requestBody:
219     x-body-name: Context
220     required: true
221     content:
```

```
      application/json:
        schema:
          type: object
          properties:
            Context:
              type: string
            Units:
              type: array
              items:
                type: object
                properties:
                  Unit:
                    type: string
                  minScale:
                    type: integer
                    format: int32
                  maxScale:
                    type: integer
                    format: int32

  responses:
    "200":
      description: "Scale created successfully"
    "400":
      $ref: "#/components/responses/BadRequest"
    "401":
      $ref: "#/components/responses/Unauthorized"
    "422":
      $ref: "#/components/responses/UnprocessableEntity"
    "500":
      $ref: "#/components/responses/InternalServerError"
  security:
    - api_key: []

ction/:
get:
  operationId: "action.read_all"
  tags:
    - "Action"
  summary: "Read the list of Actions"
  responses:
    "200":
      description: "Successfully read action list"
    "400":
      $ref: "#/components/responses/BadRequest"
    "401":
      $ref: "#/components/responses/Unauthorized"
```

```
269       "422":
270         $ref: "#/components/responses/UnprocessableEntity"
271       "500":
272         $ref: "#/components/responses/InternalServerError"
273     security:
274       - api_key: []
275
276   post:
277     operationId: "action.register"
278     tags:
279       - "Action"
280     summary: "Create a new action that the user may use for recommendation"
281
282     requestBody:
283       x-body-name: action
284       required: true
285       content:
286         application/json:
287           schema:
288             type: object
289             properties:
290               Action:
291                 type: string
292               Unit:
293                   type: string
294               Context:
295                 type: string
296               itemIds:
297                 type: array
298                 items:
299                   type: object
300                   properties:
301                     itemId:
302                       type: array
303                       items:
304                         type: object
305                         properties:
306                           user:
307                             type: string
308                           value:
309                             type: integer
310                             format: int32
311
312     responses:
313       "200":
314         description: "Action registered successfully"
315       "400":
```

```
316         $ref: "#/components/responses/BadRequest"
317       "401":
318         $ref: "#/components/responses/Unauthorized"
319       "422":
320         $ref: "#/components/responses/UnprocessableEntity"
321       "500":
322         $ref: "#/components/responses/InternalServerError"
323     security:
324       - api_key: []
325
326 ction/{Context}/{Action}/{userId}/{itemId}/{actionValue}/:
327 delete:
328     operationId: "action.delete"
329     tags:
330       - "Action"
331     summary: "Deletes an existing action"
332     parameters:
333       - $ref: "#/components/parameters/itemId"
334       - $ref: "#/components/parameters/userId"
335       - $ref: "#/components/parameters/Context"
336       - $ref: "#/components/parameters/Action"
337       - $ref: "#/components/parameters/actionValue"
338     responses:
339       "200":
340         description: "Successfully deleted context"
341       "400":
342         $ref: "#/components/responses/BadRequest"
343       "401":
344         $ref: "#/components/responses/Unauthorized"
345       "422":
346         $ref: "#/components/responses/UnprocessableEntity"
347       "500":
348         $ref: "#/components/responses/InternalServerError"
349     security:
350       - api_key: []
351
352 tem:
353  post:
354     operationId: "item.register"
355     tags:
356        - "Item"
357     summary: "Created a new item used for recommendation"
358     requestBody:
359       x-body-name: item
360       required: true
361       content:
362         application/json:
```

```
363         schema:
364           type: object
365           properties:
366             Context:
367               type: string
368             itemIds:
369               type: array
370               items:
371                 type: object
372                 required: [itemId]
373                 properties:
374                   itemId:
375                     type: string
376
377   responses:
378     "200":
379       description: "Item registered successfully"
380     "400":
381       $ref: "#/components/responses/BadRequest"
382     "401":
383       $ref: "#/components/responses/Unauthorized"
384     "422":
385       $ref: "#/components/responses/UnprocessableEntity"
386     "500":
387       $ref: "#/components/responses/InternalServerError"
388   security:
389     - api_key: []
390
391 tem/{Context}/{userId}/{itemId}:
392 delete:
393   operationId: "item.delete"
394   tags:
395     - "Item"
396   summary: "Deletes an existing item used for recommendation"
397   parameters:
398     - $ref: "#/components/parameters/itemId"
399     - $ref: "#/components/parameters/Context"
400   responses:
401     "200":
402       description: "Item deleted successfully"
403     "400":
404       $ref: "#/components/responses/BadRequest"
405     "401":
406       $ref: "#/components/responses/Unauthorized"
407     "422":
408       $ref: "#/components/responses/UnprocessableEntity"
409     "500":
```

```
410         $ref: "#/components/responses/InternalServerError"
411     security:
412       - api_key: []
413
414
415 ecommendation/:
416 get:
417     operationId: "recommendation.get_types"
418     tags:
419       - "Recommendation"
420     summary: "Returns the list of avaliable recommendation types"
421     responses:
422       "200":
423         description: "Recommendation types returned successfully"
424       "400":
425         $ref: "#/components/responses/BadRequest"
426       "401":
427         $ref: "#/components/responses/Unauthorized"
428       "422":
429         $ref: "#/components/responses/UnprocessableEntity"
430       "500":
431         $ref: "#/components/responses/InternalServerError"
432     security:
433       - api_key: []
434
435 ecommendation/{Context}/{nresult}/{rectype}/:
436 get:
437     operationId: "recommendation.get"
438     tags:
439     - "Recommendation"
440     summary: "Returns an recommendation of a given type"
441     parameters:
442       - $ref: "#/components/parameters/userId"
443       - $ref: "#/components/parameters/Action"
444       - $ref: "#/components/parameters/Context"
445       - $ref: "#/components/parameters/nresult"
446       - $ref: "#/components/parameters/rectype"
447       - $ref: "#/components/parameters/itemId"
448       - $ref: "#/components/parameters/category"
449       - $ref: "#/components/parameters/ratingtype"
450
451
452     responses:
453       "200":
454         description: "Recommendation returned successfully"
455       "400":
456         $ref: "#/components/responses/BadRequest"
```

```
    "401":
      $ref: "#/components/responses/Unauthorized"
    "422":
      $ref: "#/components/responses/UnprocessableEntity"
    "500":
      $ref: "#/components/responses/InternalServerError"
  security:
    - api_key: []

```

B.1.2 Action.py

```

action.py

rt utils
rt json
 flask import abort


read_all() -> dict:
"""Return the list of all existing actions

Returns
-------
dict
    A dict containing all the existings actions per context
"""
return utils.open_files(file="context")


register(action: dict) -> None | tuple:
CONTEXT = utils.open_files(file="context")
ACTION = utils.open_files(file="action")
if action["Context"] in CONTEXT.keys():
    if action["Context"] not in ACTION.keys():
        ACTION[action["Context"]] = {}
        ACTION[action["Context"]]["actions"] = {}
    if action["Action"] not in CONTEXT[action["Context"]]["actions"]:
        CONTEXT[action["Context"]]["actions"] = action["Action"]
        ACTION[action["Context"]]["actions"][action["Action"]] = {
            "itemID": [],
            "userID": [],
            "rating": [],
        }
        for metric in CONTEXT[action["Context"]]["Metrics"]:
```

```
35              if metric["Unit"] == action["Unit"]:
36                  ACTION[action["Context"]]["actions"][action["Action"]]["scale"] = (
37                      metric["minScale"],
38                      metric["maxScale"],
39                  )
40
41      for item in action["itemIds"][0]:
42          ACTION[action["Context"]]["actions"][action["Action"]]["itemID"].append(
43              item
44          )
45          ACTION[action["Context"]]["actions"][action["Action"]]["userID"].append(
46              action["itemIds"][0][item][0]["user"]
47          )
48          ACTION[action["Context"]]["actions"][action["Action"]]["rating"].append(
49              action["itemIds"][0][item][0]["user"]
50          )
51
52      utils.save_files("context", CONTEXT)
53      utils.save_files("action", ACTION)
54
55      # return (200, "Sucessfully created action")
56
57  else:
58      abort(422, f"Unprocessable Entity - Context dosen't exists")
59
60
61  delete(
62  Context: str, Action: str, userId: str, itemId: str, actionValue: int
63   None | tuple:
64  CONTEXT = utils.open_files(file="context")
65  ACTION = utils.open_files(file="action")
66  if Context in CONTEXT.keys():
67      if Action not in ACTION[Context]["actions"]:
68          abort(422, f"Unprocessable Entity - Action {Action} dosen't exists")
69      else:
70          ACTION[Context]["actions"].remove(Action)
71
72      utils.save_files("context", CONTEXT)
73      utils.save_files("action", ACTION)
74      return (200, "Sucessfully deleted action")
75
76  else:
77      abort(422, f"Unprocessable Entity - Context {Context} dosen't exists")
78
79
```

B.1.3 Adaptador$_{m}odelos.py$

```
1
2  rt pandas as pd
3   surprise import Dataset, NormalPredictor, Reader
4   surprise.dataset import DatasetAutoFolds
5  rt utils
6
7
8  adapt_input(
9  Context: str,
10 rectype: int,
11 nresult: int = 10,
12 Action: str = "",
13 userId: str = "",
14 itemId: str = "",
15 ratingtype: int = -1,
16 category: str = "",
17  None | dict | pd.DataFrame | DatasetAutoFolds:
18 if rectype == 0:
19     ACTION = utils.open_files(file="action")
20     df = pd.DataFrame(
21         ACTION[Context]["actions"][Action], columns=["itemID", "ratings"]
22     )
23
24     df["Ratings_per_item"] = df.groupby("itemID")["itemID"].transform("count")
25
26     output_data = pd.DataFrame(df, columns=["itemID", "Ratings_per_item"])
27
28     return output_data
29
30 if rectype == 1:
31     ACTION = utils.open_files(file="action")
32     rating_df = pd.DataFrame(
33         ACTION[Context]["actions"][Action], columns=["userID", "itemID", "rating"]
34     )
35     reader = Reader(rating_scale=(ACTION[Context]["actions"][Action]["scale"]))
36
37     data = Dataset.load_from_df(rating_df[["userID", "itemID", "rating"]], reader)
38
39     return data
40 if rectype == 2:
41     pass
42
43
44 adapt_output(
45 data: dict,
```

```
Context: str,
rectype: int,
nresult: int = 10,
Action: str = "",
userId: str = "",
itemId: str = "",
ratingtype: int = -1,
category: str = "",
 None | dict:
if rectype == 0:
    result = dict()
    result["recommendationType"] = "0"
    result["Recommendation"] = {}
    for i in data:
        result["Recommendation"][f"itemID_{i}"] = data[i]["itemID"].strip("\n")

    return result

if rectype == 1:
    return_json = {}
    return_json["recommendationType"] = "1"
    return_json["ratingType"] = ratingtype
    return_json["Recommendation"] = {}
    if Context in data["ranking"].keys():
        if Action in data["ranking"][Context].keys():
            if str(ratingtype) in data["ranking"][Context][Action].keys():
                print(userId)
                if userId in data["ranking"][Context][Action][str(ratingtype)].keys():
                    print("entrou!")
                    for item in range(
                        len(
                            data["ranking"][Context][Action][str(ratingtype)][userId][
                                :nresult
                            ]
                        )
                    ):
                        return_json["Recommendation"][f"itemID_{item}"] = data[
                        "ranking"
                    ][Context][Action][str(ratingtype)][userId][item]
                    return return_json

else:
    result = dict()
    result["recommendationType"] = "2"
    result["category"] = category
    result["Recommendation"] = {}

```

```
    if Context in data["content"]:
        if category in data["content"][Context]:
            if itemId in data["content"][Context][category]:
                for item in range(len(data["content"][Context][category][itemId])):
                    result["Recommendation"][f"itemId_{item}"] = data["content"][
                        Context
                    ][category][itemId][item]

    return result


```

B.1.4 App.py

```
p.py
rt utils
 flask import render_template  # Remove: import Flask
rt connexion
 connexion.exceptions import OAuthProblem

N_DB = utils.open_files(file="security")


apikey_auth(token, required_scopes):
info = TOKEN_DB.get(token, None)
if not info:
    raise OAuthProblem("Invalid token")

return info


= connexion.App(__name__, specification_dir="./")
add_api("../../swagger.yml")


.route("/")
home():
pass
# return render_template("home.html")


_name__ == "__main__":
app.run(host="0.0.0.0", port=8000, debug=True)
```

B.1.5 Context.py

```
1  context.py
2  rt utils
3  rt json
4   flask import Response, abort, make_response
5  rt hashlib
6
7  read_all() -> list:
8  CONTEXT = utils.open_files(file="context")
9  return list(CONTEXT.values())
10
11
12 create(Context: dict) -> None | tuple | Response:
13 CONTEXT = utils.open_files(file="context")
14 if Context["Context"] not in CONTEXT:
15     CONTEXT[Context["Context"]] = {
16         "email": Context["email"],
17         "name": Context["name"],
18         "timestamp": utils.get_timestamp(),
19         "actions": {},
20         "items": {},
21         "Metrics": [],
22         "Recommenders": {},
23         "item_schema": Context["item_schema"],
24     }
25
26     for i in Context["recommenders"][0]:
27         CONTEXT[Context["Context"]]["Recommenders"][
28             Context["recommenders"][0][i]["id"]
29         ] = Context["recommenders"][0][i]["updateTime"]
30
31     utils.save_files("context", CONTEXT)
32     TOKEN_DB = utils.open_files(file="security")
33
34     token = hashlib.md5((Context["email"] + Context["name"] +
       ↪  CONTEXT[Context["Context"]]["timestamp"]).encode('utf-8'))
35     resp = make_response(f'{Context["Context"]} created sucefully')
36     resp.headers['X-Auth-Token'] = token.hexdigest()
37     TOKEN_DB[token.hexdigest()] = Context["email"]
38     utils.save_files("security")
39     return resp
40
41 else:
42     abort(422, f"Unprocessable Entity - Context {Context} already exists")
43
44
```

```
delete(Context: str) -> None | tuple:
CONTEXT = utils.open_files(file="context")
if Context in CONTEXT:
    CONTEXT.pop(Context)
    utils.save_files("context", CONTEXT)
else:
    abort(422, f"Unprocessable Entity - Context {Context} doesn't exists")


 createScale(Context: dict) -> None | tuple:
 CONTEXT = utils.open_files(file="context")
 if Context["Context"] in CONTEXT:
     for i in Context["Units"]:
         CONTEXT[Context["Context"]]["Metrics"].append(i)


 utils.save_files("context", CONTEXT)


```

B.1.6 Item.py

```

em.py
rt utils
rt json
 flask import abort

EXT, USER = utils.open_files()
 = utils.open_files("item")


get_Itemschema() -> None | dict:
return {
    "Movie": {
        "itemId": 0000,
        "title": "title",
        "description": "description",
        "year": 0000,
        "tags": "tags",
        "director": "director",
        "actors": "actors",
    },
    "Book": {
        "itemId": 0000,
        "title": "title",
```

```
        "description": "description",
        "year": 0000,
        "tags": "tags",
        "author": "author",
    },
    "Item": {
        "itemId": 0000,
        "title": "title",
        "price": "price",
        "dateAdded": "00/00/0000",
    },
}


register(item: dict) -> None | tuple:
if item["Context"] in CONTEXT.keys():
    if item["Context"] not in ITEM.keys():
        ITEM[item["Context"]] = {}
        ITEM[item["Context"]]["items"] = {}
        ITEM[item["Context"]]["item_schema"] = CONTEXT[item["Context"]][
            "item_schema"
        ]
    schema = get_Itemschema()
    if CONTEXT[item["Context"]]["item_schema"] == "Movie":
        for i in item["itemIds"]:
            if i["itemId"] not in ITEM[item["Context"]]["items"].keys():
                ITEM[item["Context"]]["items"][i["itemId"]] = {}
            for key, value in i.items():
                if key in schema["Movie"].keys():
                    if key == "tags" and type(value) == list:
                        tags = " "
                        for i in value:
                            tags += " " + i

                        ITEM[item["Context"]]["items"][i["itemId"]]["tags"] = tags
                    else:
                        ITEM[item["Context"]]["items"][i["itemId"]][key] = value
    if CONTEXT[item["Context"]]["item_schema"] == "Book":
        for i in item["itemIds"]:
            if i["itemId"] not in ITEM[item["Context"]]["items"].keys():
                ITEM[item["Context"]]["items"][i["itemId"]] = {}
            for key, value in i.items():
                if key in schema["Book"].keys():
                    if key == "tags" and type(value) == list:
                        tags = ""
                        for v in value:
                            tags += v + ","
```

```
72
73                          tags = tags[:-1]
74
75                          ITEM[item["Context"]]["items"][i["itemId"]]["tags"] = tags
76
77                      else:
78                          ITEM[item["Context"]]["items"][i["itemId"]][key] = value
79      if CONTEXT[item["Context"]]["item_schema"] == "Item":
80          for i in item["itemIds"]:
81              if i["itemId"] not in ITEM[item["Context"]]["items"].keys():
82                  ITEM[item["Context"]]["items"][i["itemId"]] = {}
83              for key, value in i.items():
84                  if key in schema["Item"].keys():
85                      ITEM[item["Context"]]["items"][i["itemId"]][key] = value
86
87  utils.save_files("item", ITEM)
88
89
90
91  delete(Context: str, itemId: str) -> None | tuple:
92  if Context in CONTEXT.keys() and Context in ITEM.keys():
93      if itemId in ITEM[Context]["items"].keys():
94          ITEM[Context]["items"].pop(itemId)
95      else:
96          abort(422, f"Unprocessable Entity - Item {itemId} dosen't exists")
97  else:
98      abort(422, f"Unprocessable Entity - Context {Context}dosen't exists")
99
100 utils.save_files("action", ITEM)
101 return (200, "Sucessfully deleted item")
102
```

B.1.7 Recommendation.py

```
1   rt utils
2   rt json
3    flask import Response, abort, jsonify
4   rt heapq
5   rt pandas as pd
6   rt pandas.core
7    surprise import Dataset, NormalPredictor, Reader
8   rt adaptador_modelos
9
10  EXT = utils.open_files(file="context")
11  ON = utils.open_files(file="action")
12
```

```
13
14 get_types():
15 return {
16     "rectypes": {
17         0: "Popularity",
18         1: {"Ranking": {0: "SVD", 1: "CoClustering", 2: "KNNBasic"}},
19         2: "Content-based",
20     }
21 }
22
23
24 get(
25 Context: str,
26 Action: str,
27 nresult: int,
28 rectype: int,
29 ratingtype: int = 0,
30 category: str = "",
31 userId: str = "",
32 itemId: str = "",
33  None | dict | str:
34 if rectype == 0:
35     return get_popRecommendation(Context, nresult, Action)
36
37 elif rectype == 1:
38     return get_ratingRecommendation(Context, nresult, Action, userId, ratingtype)
39
40 else:
41     return get_contentBased(Context, itemId, nresult, category)
42
43
44 get_popRecommendation(Context: str, nresult: int, Action: str) -> None | str:
45 rating_count = adaptador_modelos.adapt_input(Context, 0, nresult, Action)
46 rating_count.sort_values(
47     "Ratings_per_item", ascending=False
48 ).drop_duplicates().head(nresult)
49
50 tmp_result = rating_count.to_dict(orient="index")
51 return_json = adaptador_modelos.adapt_output(tmp_result, Context, 0, nresult)
52 return_json = json.dumps(return_json)
53 return return_json
54
55
56 get_ratingRecommendation(
57 Context: str, nresult: int, Action: str, userId: str, ratingtype: int
58  None | dict:
59 print(Context)
```

```
RECOMMENDATION = utils.open_files("recommendation")
return_json = adaptador_modelos.adapt_output(
    RECOMMENDATION, Context, 1, Action=Action, userId=userId, ratingtype=ratingtype
)
print(return_json)
return return_json


get_contentBased(
Context: str, itemId: str, nresult: int, category: str
 None | dict:
RECOMMENDATION = utils.open_files("recommendation")
return_json = adaptador_modelos.adapt_output(
    RECOMMENDATION, Context, 2, itemId=itemId, nresult=nresult, category=category
)

return return_json

```

B.1.8 Recommendation$_{t}ools.py$

```

 collections import defaultdict
rt json
 surprise import CoClustering, Dataset, SVD, KNNBasic, accuracy
rt pandas as pd
rt pandas.core
 surprise import Dataset, NormalPredictor, Reader
 surprise.prediction_algorithms import predictions
 surprise.model_selection import KFold
rt utils
 sklearn.metrics.pairwise import linear_kernel
 sklearn.feature_extraction.text import TfidfVectorizer
rt adaptador_modelos

MMENDATION = dict()
MMENDATION["ranking"] = {}
MMENDATION["content"] = {}


get_top_n(predictions: list, n=10) -> defaultdict:
"""Return the top-N recommendation for each user from a set of predictions.

Args:
    predictions(list of Prediction objects): The list of predictions, as
        returned by the test method of an algorithm.
```

```
    n(int): The number of recommendation to output for each user. Default
        is 10.

Returns:
A dict where keys are user (raw) ids and values are lists of tuples:
    [(raw item id, rating estimation), ...] of size n.
"""

# First map the predictions to each user.
top_n = defaultdict(list)
for uid, iid, true_r, est, _ in predictions:
    top_n[uid].append((iid, est))

# Then sort the predictions for each user and retrieve the k highest ones.
for uid, user_ratings in top_n.items():
    user_ratings.sort(key=lambda x: x[1], reverse=True)
    top_n[uid] = user_ratings[:n]

return top_n




get_all_top_n() -> None:
ACTION = utils.open_files("action")
for c in ACTION.keys():
    RECOMMENDATION["ranking"][c] = {}
    for a in ACTION[c]["actions"]:

        RECOMMENDATION["ranking"][c][a] = {}
        RECOMMENDATION["ranking"][c][a][0] = {}
        data = adaptador_modelos.adapt_input(c, 1, Action=a)

        # SDV
        trainset = data.build_full_trainset()  # DIVIDIR AQUI
        algo = SVD()
        algo.fit(trainset)

        testset = trainset.build_anti_testset()
        predictions = algo.test(testset)
        RECOMMENDATION["ranking"][c][a][0]["rmse"] = accuracy.rmse(predictions,
        ↪  verbose=True)

        top_n = get_top_n(predictions, n=10)
        for uid, user_ratings in top_n.items():
            RECOMMENDATION["ranking"][c][a][0][uid] = [
                iid for (iid, _) in user_ratings
```

```
72              ]
73
74          # CoCLuresting
75
76          RECOMMENDATION["ranking"][c][a][1] = {}
77          trainset = data.build_full_trainset()
78          algo = CoClustering()
79          algo.fit(trainset)
80
81          testset = trainset.build_anti_testset()
82          predictions = algo.test(testset)
83          RECOMMENDATION["ranking"][c][a][1]["rmse"] = accuracy.rmse(predictions,
            ↪  verbose=True)
84
85          top_n = get_top_n(predictions, n=10)
86          for uid, user_ratings in top_n.items():
87              RECOMMENDATION["ranking"][c][a][1][uid] = [
88                  iid for (iid, _) in user_ratings
89              ]
90
91          # KNNBasic
92          RECOMMENDATION["ranking"][c][a][2] = {}
93          trainset = data.build_full_trainset()
94          algo = KNNBasic()
95          algo.fit(trainset)
96
97          testset = trainset.build_anti_testset()
98          predictions = algo.test(testset)
99          RECOMMENDATION["ranking"][c][a][2]["rmse"] = accuracy.rmse(predictions,
            ↪  verbose=True)
100         top_n = get_top_n(predictions, n=10)
101         for uid, user_ratings in top_n.items():
102             RECOMMENDATION["ranking"][c][a][2][uid] = [
103                 iid for (iid, _) in user_ratings
104             ]
105
106 utils.save_files("recommendation", RECOMMENDATION)
107
108 content_based():
109 ITEM = utils.open_files("item")
110 for context in ITEM.keys():
111     RECOMMENDATION["content"][context] = {}
112     data = pd.DataFrame.from_dict(ITEM[context]["items"], orient="index")
113     if ITEM[context]["item_schema"] == "Movie":
114         for key, _ in data.items():
115             if key in [
116                 "title",
```

```
                "description",
                "year",
                "tags",
                "director",
                "actors",
            ]:
                # Remove palavras de parada, como "the", "and"
                data_tfidf = TfidfVectorizer(stop_words="english")
                data[key] = data[key].fillna("")
                data_key_matrix = data_tfidf.fit_transform(
                    data[key]
                )  # Gera a matrix para calculo da similaridade do coseno

                # print(data_key_matrix.shape)

                similaridade_coseno = linear_kernel(
                    data_key_matrix, data_key_matrix
                )
                RECOMMENDATION["content"][context][key] = {}
                for idx in range(data.shape[0]):
                    RECOMMENDATION["content"][context][key][
                        data["itemId"].iloc[idx]
                    ] = []
                    scores = list(enumerate(similaridade_coseno[idx]))
                    scores = sorted(scores, key=lambda x: x[1], reverse=True)
                    scores = scores[1:10]
                    index = [i[0] for i in similaridade_coseno]
                    for p in data["itemId"].iloc[index]:
                        RECOMMENDATION["content"][context][key][
                            data["itemId"].iloc[idx]
                        ].append(p)

    if ITEM[context]["item_schema"] == "Book":
        for key, _ in data.items():
            if key in ["title", "description", "year", "tags", "author"]:
                # Remove palavras de parada, como "the", "and"
                data_tfidf = TfidfVectorizer(stop_words="english")
                data[key] = data[key].fillna("")
                data_key_matrix = data_tfidf.fit_transform(
                    data[key]
                )  # Gera a matrix para calculo da similaridade do coseno

                # print(data_key_matrix.shape)

                similaridade_coseno = linear_kernel(
                    data_key_matrix, data_key_matrix
                )
```

```
                RECOMMENDATION["content"][context][key] = {}
                for idx in range(data.shape[0]):
                    RECOMMENDATION["content"][context][key][
                        data["itemId"].iloc[idx]
                    ] = []
                    scores = list(enumerate(similaridade_coseno[idx]))
                    scores = sorted(scores, key=lambda x: x[1], reverse=True)
                    scores = scores[1:10]
                    index = [i[0] for i in similaridade_coseno]
                    for p in data["itemId"].iloc[index]:
                        RECOMMENDATION["content"][context][key][
                            data["itemId"].iloc[idx]
                        ].append(p)

    if ITEM[context]["item_schema"] == "Item":
        for key, _ in data.items():
            if key in ["title", "price", "dataAdded"]:
                # Remove palavras de parada, como "the", "and"
                data_tfidf = TfidfVectorizer(stop_words="english")
                data[key] = data[key].fillna("")

                data_key_matrix = data_tfidf.fit_transform(
                    data[key]
                )  # Gera a matrix para calculo da similaridade do coseno

                # print(data_key_matrix.shape)

                similaridade_coseno = linear_kernel(
                    data_key_matrix, data_key_matrix
                )
                RECOMMENDATION["content"][context][key] = {}
                for idx in range(data.shape[0]):
                    RECOMMENDATION["content"][context][key][
                        data["itemId"].iloc[idx]
                    ] = []
                    scores = list(enumerate(similaridade_coseno[idx]))
                    scores = sorted(scores, key=lambda x: x[1], reverse=True)
                    scores = scores[1:10]
                    index = [i[0] for i in similaridade_coseno]
                    for p in data["itemId"].iloc[index]:
                        RECOMMENDATION["content"][context][key][
                            data["itemId"].iloc[idx]
                        ].append(p)

utils.save_files("recommendation", RECOMMENDATION)


```

```
211
212 precision_recall_at_k(predictions, k=10, threshold=3.5):
213 """Return precision and recall at k metrics for each user"""
214
215 # First map the predictions to each user.
216 user_est_true = defaultdict(list)
217 for uid, _, true_r, est, _ in predictions:
218     user_est_true[uid].append((est, true_r))
219
220 precisions = dict()
221 recalls = dict()
222 for uid, user_ratings in user_est_true.items():
223
224     # Sort user ratings by estimated value
225     user_ratings.sort(key=lambda x: x[0], reverse=True)
226
227     # Number of relevant items
228     n_rel = sum((true_r >= threshold) for (_, true_r) in user_ratings)
229
230     # Number of recommended items in top k
231     n_rec_k = sum((est >= threshold) for (est, _) in user_ratings[:k])
232
233     # Number of relevant and recommended items in top k
234     n_rel_and_rec_k = sum(
235         ((true_r >= threshold) and (est >= threshold))
236         for (est, true_r) in user_ratings[:k]
237     )
238
239     # Precision@K: Proportion of recommended items that are relevant
240     # When n_rec_k is 0, Precision is undefined. We here set it to 0.
241
242     precisions[uid] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0 else 0
243
244     # Recall@K: Proportion of relevant items that are recommended
245     # When n_rel is 0, Recall is undefined. We here set it to 0.
246
247     recalls[uid] = n_rel_and_rec_k / n_rel if n_rel != 0 else 0
248
249 return precisions, recalls
250
251 get_recall():
252 ACTION = utils.open_files("action")
253 for c in ACTION.keys():
254     RECOMMENDATION["ranking"][c] = {}
255     for a in ACTION[c]["actions"]:
256
257         RECOMMENDATION["ranking"][c][a] = {}
```

```
258         RECOMMENDATION["ranking"][c][a][0] = {}
259         data = adaptador_modelos.adapt_input(c, 1, Action=a)
260 
261         # SDV
262         trainset = data.build_full_trainset()  # DIVIDIR AQUI
263         kf = KFold(n_splits=5)
264         algo = SVD()
265 
266         for trainset, testset in kf.split(data):
267             algo.fit(trainset)
268             predictions = algo.test(testset)
269             precisions, recalls = precision_recall_at_k(predictions, k=5, threshold=4)
270 
271             # Precision and recall can then be averaged over all users
272             print(sum(prec for prec in precisions.values()) / len(precisions))
273             print(sum(rec for rec in recalls.values()) / len(recalls))
274 
275             precision_t = sum(prec for prec in precisions.values()) / len(precisions)
276             recall_t = sum(rec for rec in recalls.values()) / len(recalls)
277             f1_score = 2 * ((precision_t * recall_t )/ (precision_t+recall_t))
278             print(f"f1={f1_score}")
279 
280 
281 all_top_n()
282 ent_based()
283 recall()
```

B.2 Arquivos JSON apresentados

B.2.1 Registro de Ações

B.2.2 Criação de Métrica

B.2.3 Exemplo de descriação de filme

```
ction": "Rating",
nit": "5-stars",
ontext_id": "BLPL",
temIds": [
{
  "item0001": [
    {
      "user": "user0001",
      "ParameterValue": 4
    },
    {
      "user": "user0002",
      "ParameterValue": 7
    },
    {
      "user": "user0003",
      "ParameterValue": 5
    }
  ],
  "item0002": [
    {
      "user": "user0002",
      "ParameterValue": 4
    },
    {
      "user": "user0004",
      "ParameterValue": 1
    }
  ]
}
```

```
"Unit": "5-stars",
"maxScale": 1,
"minScale": 5
```

```
"id": "1274",
"title": "Akira",
"description": "In 2019, following a world war triggered by ...",
"directors": ["Katsuhiro Otomo"],
"actors": [
    "Mitsuo Iwata",
    "Nozomu Sasaki",
    "Mami Koyama"
],
"genres": [
    "Adventure",
    "Animation",
    "Sci-Fi",
    "Thriller"
]
```

B.2.4 Exemplo de configuração de recomendadores

```
"id": "Popularity",
"updateTime": "00000", \\ 0 minutes
"parameters": {
  "actions": [
    "view",
    "buy"
  ]
}


"id": "SVD",
"updateTime": "00360", \\6 hours
"parameters": {
  "n_factors": 300,
  "n_epochs": 10,
  "biased": false
}
```

B.2.5 Exemplo de uma lista de recomendações para um certo usuário

```
{
    "ranking":{
        "100k-ml":{
            "rating":{
                "SDV":
                    {
                        "151": [      // User Id
                            "661",
                            "914",
                            "3408",
                            "2355",
                            "1197",
                            "1287",
                            "2804",
                            "594",
                            "919",
                            "595"
                        ]
                    }
            }
        }

    }
}
```

B.2.6 Exemplo de uma lista de recomendação para um certo item

```
{
    "item":{
        "100k-ml":{
            "content-based":{
                "genre":
                    {
                        "1274": [      // Item Id
                            "741",
                            "1025",
                            "1151",
                            "1205",
                            "2137",
                            "1287",
                            "783",
                            "334",
                            "664",
                            "1593"
                        ]
                    }
            }
        }

    }
}
```

B.2.7 Exemplo de requisição para o registro de um contexto

```
$ curl -X 'POST' \
    'http://127.0.0.1:8000/api/context' \
    -H 'accept: */*' \
    -H 'Content-Type: application/json' \
    -d '{
        "context_id": "BLPL",
        "url": "https://www.literaturabrasileira.ufsc.br",
        "email": "hans.buss@mailfence.com",
        "name": "Hans Buss",
        "recset": [
            {
                "id": "Popularity",
                "updateTime": "0s",
                "parameters": {
                    "actions": [
                        "view",
                        "buy"
                    ]
                }
            },
            {
                "id": "SVD",
                "updateTime": "6h",
                "parameters": {
                    "n_factors": 300,
                    "n_epochs": 10,
                    "biased": false
                }
            }
        ],
        "units": [
            {
                "Unit": "5-stars",
                "maxScale": 1,
                "minScale": 5
            }
        ]
    }'
```

```
$ curl -X 'DELETE' \
    'http://127.0.0.1:8000/api/context/BLPL' \
    -H 'accept: */*' \
    -H 'X-Auth: CONTEXTKEY'
```

B.2.8 Exemplo de resposta positiva a requisição de registro de um contexto

```
$HTTP/1.1 200 OK
Content-Type: application/json
Date: Mon, 28 Dez 2023 14:09:34 GMT
Content-Length: 3
Connection: keep-alive'
x-auth-token: 7a14b8beb4cdcc2d01e3b68d828cd08c

{

    "X-Auth-Token": '7a14b8beb4cdcc2d01e3b68d828cd08c'
}
```

B.2.9 Exemplo de requisição para deleção de um contexto

B.2.10 Exemplo de uma requisição para registro de ações

```
curl -X 'POST' \
    'http://127.0.0.1:8000/api/action/' \
    -H 'accept: */*' \
    -H 'X-Auth: CONTEXTKEY' \
    -H 'Content-Type: application/json' \
    -d '{
       "context_id": "BLPL",
       "actions": [
       {
            "Action": "Rating",
            "Unit": "5-stars",
            "Context_id": "BLPL",
            "itemIds": [
                {
                    "item0001": [
                    {
                        "user": "user0001",
                        "ParameterValue": 4
                    },
                    {
                        "user": "user0002",
                        "ParameterValue": 7
                    },
                    {
                        "user": "user0003",
                        "ParameterValue": 5
                    }
                    ],                                          "item0002": [
                    {
                        "user": "user0001",
                        "ParameterValue": 4
                    },
                    {
                        "user": "user0004",
                        "ParameterValue": 1
                    }
                ]
            }
        ]
    }
]
}
```

B.2.11 Exemplo de uma requisição para registro de itens

```
            curl -X 'POST' \
                'http://127.0.0.1:8000/api/item/registerItem/' \
                -H 'accept: */*' \
                -H 'X-Auth: CONTEXTKEY' \
                -H 'Content-Type: application/json' \
                -H 'Content-Type: application/json' \
                -d '{
"context_id": "BLPL",
"itensSet": [
    {
        "id": "item0001",
        "title": "Akira",
        "description": "In 2019, following a world war triggered by ...",
        "directors": [
            "Katsuhiro Otomo"
        ],
        "actors": [
            "Mitsuo Iwata",
            "Nozomu Sasaki",
            "Mami Koyama"
        ],
        "genres": [
            "Adventure",
            "Animation",
            "Sci-Fi",
            "Thriller"
        ]
    }
]
```

B.2.12 Exemplo de uma requisição para remoção de um certo usuário

```
            'http://127.0.0.1:8000/api/item/BLPL/user0001/' \
             -H 'accept: */*' \
             -H 'X-Auth: CONTEXTKEY'
```

B.2.13 Exemplo de retorno de um pedido de recomendação

```
{
    "recommendationType": "SVD",
    "recommendationId": 1,
    "Recommendation": {

        "item0002": "0.987",
        "item0454": "0.982",
        "item0434": "0.972"
        }
  }
```

B.2.14 Exemplo de uma requisição de um pedido de recomendação

```
$ curl -X 'GET' \
    'http://127.0.0.1:8000/api/recommendation/BDLP/SVD/user0001/10
    /BLPL/userId
    /0/0' \
    -H 'accept: */*' \
    -H 'X-Auth: CONTEXTKEY'
```