Ministério da Educação
Secretaria de Educação Profissional e Tecnológica
Instituto Federal Catarinense
Câmpus Videira

Iury Krieger

UMA API WEB ORIENTADA A METADADOS COMO SERVIÇO DE RECOMENDAÇÃO HÍBRIDA

Orientador: Msc. Tiago Heineck

Coorientador: Msc. Wanderson Rigo

Videira - Santa Catarina

2017

Ministério da Educação
Secretaria de Educação Profissional e Tecnológica
Instituto Federal Catarinense
Câmpus Videira

Iury Krieger

UMA API WEB ORIENTADA A METADADOS COMO SERVIÇO DE RECOMENDAÇÃO HÍBRIDA

Trabalho de conclusão de curso submetido ao Instituto Federal Catarinense - Campus Videira como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Msc. Tiago Heineck

Coorientador: Msc. Wanderson Rigo

Videira - Santa Catarina
2017

Iury Krieger

UMA API WEB ORIENTADA A METADADOS COMO SERVIÇO DE RECOMENDAÇÃO HÍBRIDA

Trabalho de conclusão de curso submetido ao Instituto Federal Catarinense - Campus Videira como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Videira (SC), 16 de Maio de 2017

Msc. Tiago Heineck
Instituto Federal Catarinense

Msc. Wanderson Rigo
Instituto Federal Catarinense

BANCA EXAMINADORA

Msc. Marcelo Cendron
Instituto Federal Catarinense

Maurício Ferreira
Instituto Federal Catarinense

Resumo

Devido a expansão massiva de dados produzidos e disponíveis na Internet, os usuários estão cada vez mais sobrecarregados de informação, não sabendo distinguir informações realmente úteis. Para sanar este problema, os sistemas de recomendação visam recomendar os itens mais úteis a cada usuário, através de técnicas de machine learning. Tais técnicas visam prever a avaliação de um usuário a um item, baseando-se nas avaliações já conhecidas. Este trabalho propõe o desenvolvimento de uma API Web de código aberto que recomenda itens a usuários, fazendo uso de um sistema de recomendação híbrido que analisa as estruturas pré definidas e proporciona recomendações, baseando-se nos metadados fornecidos, através do conteúdo do item e da filtragem colaborativa de usuários. Tal sistema poderá processar suas recomendações utilizando a GPU, minimizando o tempo da requisição de recomendação e consequentemente aumentando a eficiência da aplicação. Dessa forma é possível fornecer um serviço multipropósito desprendido de qualquer ambiente e linguagem de programação, trazendo uma visão mais transparente dos sistemas de recomendação aos desenvolvedores.

Palavras-chaves: Sistemas de Recomendação. Aprendizado de Máquina. Metadados. Computação em GPU.

Abstract

Due to the massive expansion of data produced and available on the Internet, users are increasingly overloaded with information, not knowing how to distinguish which is really useful. To remedy this problem, recommendation systems aim to recommend the most useful items to each user through machine learning techniques. These techniques are intended to predict a user's rating of an item, based on previously known rating. This work proposes the development of an open-source Web API that recommends items to users, making use of a hybrid recommendation system that analyzes the pre-defined structures and provides recommendations, based on the metadata provided, through item content and collaborative filtering. Such a system can process its recommendations using the GPU, minimizing the time of the recommendation request and consequently increasing the application efficiency. Therefore is possible to provide a multi-purpose service detached from any environment and programming language, bringing a more transparent view of recommendation systems to developers.

key-words: Recommender Systems. Machine Learning. Metadata. GPU Computing.

Lista de Quadros

Lista de ilustrações

Lista de abreviaturas e siglas

IA	*Inteligência Artificial*
HTTP	*Hypertext Transfer Protocol*
API	*Application Program Interface*
REST	*Representational State Transfer*
JSON	*Javasript Object Notation*

Sumário

1 INTRODUÇÃO

Com o avanço crescente do campo tecnológico, os computadores vêm desempenhando tarefas antes incumbidas à seres humanos. O poder de computação provou-se muito eficaz ao desempenhar tarefas que possuíssem um padrão possível de se expressar através de um algoritmo, mais ainda, se este padrão fosse repetitivo.

Logo os computadores começaram a desempenhar funções nas mais diversas áreas, desde cálculos matemáticos à manipulação de imagens. Atualmente, das funções desempenhadas pelos computadores, a mais difícil de se reproduzir com precisão é o padrão de raciocínio humano.

Alguns autores defendem que para que um computador atinja tal nível, seria necessário que o mesmo possuísse consciência, assim como os seres humanos. Outros defendem que o raciocínio humano não consegue ser reproduzido, apenas emulado, devido à impossibilidade de se programar uma consciência computacional. Tal área de estudo, que tem como o foco o desenvolvimento de sistemas computacionais rumo a proximidade do método humano, chama-se inteligência artificial (RUSSELL; NORVIG, 2004; COPPIN, 2015).

Possível ou não, é inegável o avanço da inteligência artificial desde seu início nos primórdios da computação. Algumas tarefas, tais como a atribuição de uma consciência a um sistema computacional, deixaram de ser o foco da área, uma vez que não possuímos a tecnologia para construir sistemas muito mais complexos que os atuais (RUSSELL; NORVIG, 2004).

Entretanto, a inteligência artificial encontrou-se muito eficaz em outras áreas do método humano, tais como o aprendizado, um dos segmentos mais importantes da área, dentro da inteligência artificial chamado de aprendizado de máquina (*machine learning*) (COPPIN, 2015).

Desde os anos 90 a preocupação com o armazenamento e a expansão massiva de dados produzidos já existia, prevendo que usuários ficariam sobrecarregados de informação, não sabendo distinguir o que seria realmente útil (HILL *et al.*, 1995; ADOMAVICIUS; TUZHILIN, 2005). Na época, uma comunidade virtual de avaliação foi proposta para proporcionar aos usuários o mínimo de esforço ao encontrar informações úteis. Com a evolução da inteligência artificial e das técnicas de machine learning, este trabalho de avaliação e recomendação, antes feito por uma comunidade, hoje é atribuído aos sistemas de recomendação (HILL *et al.*, 1995).

Sistemas de recomendação (RSs) são ferramentas de software e técnicas que provém

sugestões de artefatos à usuários. Estes artefatos são definidos como os objetos de valor à serem recomendados (RICCI; ROKACH; SHAPIRA, 2011). Atualmente, o interesse em tais sistemas se mantém alto, devido a abundância de aplicações práticas (ADOMAVICIUS; TUZHILIN, 2005), exemplificadas nos casos de *E-commerce* por Schafer, Konstan e Riedl (2001), além de Linden, Smith e York (2003), onde são amplamente utilizados.

Desta forma, sistemas de recomendação vem sendo desenvolvidos para a resolução do problema descrito nas mais diversas áreas (BENNETT; LANNING *et al.*, 2007; GAVALAS *et al.*, 2014), desde aplicações hoteleiras como o TripAdvisor até aplicações de entretenimento como a Netflix, além da sua origem nos *E-commerces* citados anteriormente. Muitos destes sistemas são casos de RSs aplicados a itens e finalidades específicas (HUANG *et al.*, 2002; BROZOVSKY; PETRICEK, 2007), onde todo o motor de recomendação segue uma abordagem baseada no padrão que lhe foi dado.

Por outro lado, ao observar aplicações web de sistemas de recomendação, verifica-se a existência de soluções em forma de APIs, tais como o Google Cloud Platform e o Microsoft Cognitive Services, fornecidas como serviços transparentes. Entretanto, estas soluções proprietárias não são incorporadas a aplicação, mas sim utilizadas como serviços externos, dificultando a personalização.

Para sanar estes problemas, este trabalho propõe o desenvolvimento de uma API que proporcione uma visão mais transparente dos sistemas de recomendação, permitindo ao usuário desfrutar das funcionalidades, sem a necessidade de um profundo conhecimento dos detalhes que compõem as diferentes técnicas de recomendação, além dos problemas decorrentes do uso de cada uma das técnicas. Além disso, tal tecnologia será fornecida como um serviço de código aberto, podendo ser utilizada em qualquer ambiente.

Este trabalho está dividido em seis seções. A segunda seção apresenta o referencial teórico necessário para o entendimento total do escopo do trabalho. Na terceira seção são apresentadas as principais características do trabalho proposto, além de compará-lo com outros trabalhos relacionados. Em seguida, a quarta seção apresenta a metodologia a ser utilizada para realização do trabalho proposto na seção anterior. Mais à frente, na seção cinco, será abordado o cronograma a ser empregado para a realização do trabalho e, por fim, na sexta seção são apresentadas as considerações finais.

1 Objetivos

1.1 Objetivo Geral

Desenvolver uma API web de código aberto para recomendação híbrida de itens a usuários.

1.2 Objetivos Específicos

- Fornecer uma documentação das funcionalidades visando futura colaboração da comunidade e utilização por outros desenvolvedores.
- Proporcionar a recomendação das propriedades relevantes através das estruturas de metadados fornecidas.

2 Metodologia

A metodologia deste trabalho está dividida em três seções. Primeiramente serão implementadas todas as funcionalidades descritas anteriormente. Mais à frente, será feita a validação das funcionalidades implementadas e da eficácia das recomendações. Por fim, serão feitos os ajustes e correções necessárias de acordo com o resultado da validação das funcionalidades implementadas.

2.1 Implementação

Inicialmente serão implementados os algoritmos de recomendação híbrida, incluindo o processamento dos metadados fornecidos. Os algoritmos de recomendação resumem a eficácia da API e devem consumir a maior parte do tempo de desenvolvimento.

Ao completar a implementação das técnicas híbridas de sistemas de recomendação, serão implementadas as demais funcionalidades da API. Serão consideradas a identificação e entrada dos metadados, além do formato dos dados de saída.

Ao fim do desenvolvimento, o código será adaptado para processamento na GPU através de diretivas de compilação e bibliotecas especializadas.

2.2 Validação

Assim que a API esteja em um grau considerado funcional, será feita a validação da eficácia ao recomendar as estruturas fornecidas através de grupos de testes definidos, uma técnica amplamente utilizada na validação de técnicas de machine learning.

A validação será feita utilizando um grupo separado dos dados utilizados para testes, confrontando as recomendações feitas com o resultado esperado. Através desses resultados é medida a acurácia de um sistema de recomendação, métrica utilizada como medida de eficiência entre os diferentes métodos utilizados.

2.3 Ajustes e Correções

Por fim, serão feitos os ajustes e correções de erros recolhidos ao longo do processo, além de testar as funcionalidades e a utilização da API como um todo. A documentação será feita durante boa parte de todo o processo e, neste caso em específico, possui um foco especial, uma vez que o princípio da API é que a mesma seja utilizável por outros desenvolvedores, além de possibilitar a contribuição da comunidade.

3 Trabalhos Relacionados

Tendo como base as técnicas descritas acima, existem trabalhos como os apresentados por Guo *et al.* (2015), que abordam as técnicas em forma de biblioteca Java a ser incluída nos projetos. Esta abordagem torna a utilização mais simples devido ao fato do usuário poder utilizar apenas as funcionalidades da biblioteca, preocupando-se com o formato de entrada e saída dos dados, não com o processo de recomendação em si. Outra abordagem interessante é a proposta por Brozovsky e Petricek (2007) ao construir uma biblioteca C# multipropósito, focando na recomendação de itens com base na avaliação em um esquema de *rating* (de uma a cinco estrelas), ou com base apenas em itens com avaliação positiva.

Do mesmo modo que a biblioteca desenvolvida por Brozovsky e Petricek (2007), a API proposta neste trabalho também visa ser multipropósito e distribuída como código aberto pela licença pública GNU (**GPL**), porém, fornecendo tais funcionalidades como um serviço web independente de linguagem de programação, o que não acontece nos exemplos apresentados.

Além dos trabalhos apresentados, Nascimento (2013) aborda os sistemas de recomendação com uma perspectiva semelhante a este trabalho, focando mais no ganho de desempenho ao processar o método de filtragem colaborativa na GPU. Este trabalho não tem seu foco em desempenho, mas sim em uma proposta de **recomendação genérica**, que forneça recomendações a quaisquer modelos de usuários e itens através do método híbrido.

2 REFERENCIAL TEÓRICO

1 Aprendizado de Máquina

Um dos segmentos da inteligência artificial com grande importância na atualidade é o aprendizado de máquina. Responsável pela construção de agentes capazes de, a partir de uma coleção de pares de entrada e saída, aprender uma função que prevê a saída para novas entradas. Tais agentes são definidos como tudo que pode perceber seu ambiente através de sensores, além de atuar sobre o mesmo através de atuadores. Em outras palavras, o aprendizado de máquina resume-se em técnicas que proporcionam a um algoritmo a capacidade de melhorar seu desempenho de forma automática, através do conhecimento obtido pelas entradas existentes (COPPIN, 2015).

Dessa forma, considera-se que um agente está aprendendo se melhorar o seu desempenho nas tarefas para que foi designado, a partir de suas observações sobre o mundo. Este aprendizado proporciona às técnicas de *machine learning* a capacidade evolutiva, uma vez que é possível não só responder as entradas do mundo exterior como também tirar conclusões sobre as mesmas, melhorando cada vez mais a natureza da solução (RUSSELL; NORVIG, 2004).

Conforme apresentado por Carbonell, Michalski e Mitchell (1983), devido a capacidade de, além de solucionar problemas, melhorar automaticamente o desempenho da solução, os sistemas de aprendizagem tem suas aplicações nas mais diversas áreas, tais como agricultura, educação, sistemas especialistas de alta performance, reconhecimento de imagem, programação, etc. Através de um apanhado das aplicações nas áreas de utilização, Carbonell, Michalski e Mitchell (1983) dividem o campo de aprendizado do *machine learning* em três partes:

- **Estudos orientados à tarefa (*Task-oriented studies*):** composto pelo desenvolvimento e análise de sistemas de aprendizagem visando melhorar a performance na solução de determinadas tarefas.
- **Simulação cognitiva (*Cognitive simulation*):** formado pela investigação e simulação do processo de aprendizagem humano.
- **Análise teórica (*Theoretical analysis*):** exploração teórica do espaço de possíveis processos de aprendizado.

Analisando a taxonomia proposta por Carbonell, Michalski e Mitchell (1983), pode-se identificar que o escopo deste trabalho encontra-se nos estudos orientados à tarefa, onde

o propósito é a melhoria da performance, neste caso através de recomendações orientadas à metadados.

Como exemplo do uso das técnicas de *machine learning*, Sebastiani (2002) apresenta um algoritmo de categorização de texto que, a partir de um conjunto de documentos pré-classificados (entradas), constrói um classificador para novos documentos (novas entradas). Outro exemplo, apresentado por Pang, Lee e Vaithyanathan (2002), reforça a ideia de melhora de desempenho para novas entradas através de um padrão aprendido a partir de entradas já existentes. Através de dados sobre avaliações de filmes, pode-se perceber que, mesmo as técnicas padrão de *machine learning*, acabam superando os patamares humanos na classificação de sentimentos.

2 Sistemas de Recomendação

Como ramificação do aprendizado de máquina, os sistemas de recomendação (RSs) são técnicas de software que provém sugestões a usuários de itens que os mesmos possam querer utilizar (RESNICK; VARIAN, 1997; SCHAFER; KONSTAN; RIEDL, 1999). Desta forma, recomendações seriam, em sua forma mais simples, rankings de itens, tais como os utilizados na maioria das soluções de produtos (livros mais lidos, filmes mais assistidos, etc.) (RICCI; ROKACH; SHAPIRA, 2011). O que os RSs trazem de novo é a tentativa de predizer, através da filtragem colaborativa ou da similaridade de conteúdo, qual o ranking mais adequado de produtos ou serviços a um usuário. A filtragem colaborativa, termo cunhado por Resnick e Varian (1997), recomenda itens baseando-se nos relacionamentos do usuário. Por outro lado, a similaridade de conteúdo baseia-se no conteúdo de itens já avaliados pelo usuário. Tais dados podem ser coletadas de forma explícita, na forma de perguntas diretas e avaliações do usuário sobre os itens, ou de forma interpretativa, inferindo sobre ações tomadas pelo usuário e atribuindo peso a elas.

Mais formalmente, os sistemas de recomendação podem ser descritos matematicamente da seguinte forma: sendo C o conjunto de todos os usuários e S o conjunto de todos os itens que podem ser recomendados, tanto o espaço S como o espaço C podem ser extremamente grandes, batendo os milhões de usuários e itens (ADOMAVICIUS; TUZHILIN, 2005; GOMEZ-URIBE; HUNT, 2016). Dessa forma, tem-se u como a função de utilidade de um item s para um usuário c. A função u utiliza-se do conjunto ordenado R, descrito como $C \times S \rightarrow R$, para encontrar o item $s \in S$ com a maior utilidade para o usuário c. Um exemplo de como as preferências são armazenadas no espaço de avaliações C x S pode ser visto no quadro 1.

De acordo com o quadro 1, o símbolo "$\emptyset$"representa os filmes ainda não avaliados pelos usuários. Estes itens, por sua vez, são os alvos das técnicas de recomendação que tentam predizer a avaliação de um usuário. Uma vez que o motor de recomendação pode

Quadro 1 – Exemplo de matriz de recomendações a filmes

	K-PAX	Life of Brian	Memento	Notorious
Alice	4	3	2	4
Bob	∅	4	5	5
Cindy	2	2	4	∅
David	3	∅	5	2

Fonte: Adomavicius e Tuzhilin (2005)

predizer as avaliações de um usuário, pode-se recomendar ao mesmo apenas os N itens com a maior avaliação estimada (ADOMAVICIUS; TUZHILIN, 2005).

Como consequência da importante participação dos sistemas de recomendação em sites com um grande número de público, tais como Netflix, eBay e Amazon.com, os mesmos tornaram-se ferramentas poderosas (SCHAFER; KONSTAN; RIEDL, 1999) e são considerados os propulsores de várias estatísticas, entre elas: o aumento da satisfação dos usuários, devido a precisão das recomendações; o aumento da fidelidade dos usuários, devido ao aumento de precisão quanto maior for a interação do usuário com o site; a aumento da capacidade do próprio serviço em entender melhor as intenções de seu público (RICCI; ROKACH; SHAPIRA, 2011). Tendo em vista o crescimento do número de aplicações que utilizam sistemas de recomendação e da variedade de soluções utilizadas em grandes sites, torna-se notável a importância dos mesmos.

Como próximo passo na evolução dos sistemas de recomendação, Adomavicius e Tuzhilin (2015) propõem que os RSs, além de considerarem a similaridade entre perfis, devem estar cientes do contexto da avaliação do usuário ao construírem o modelo de perfil. Chamados de sistemas cientes de contexto, estes sistemas de recomendação devem diferenciar a ação que o usuário toma ao apenas analisar um item (filme, produto, etc.), não necessariamente indicando que itens parecidos devem ser recomendados no futuro, da ação tomada ao consumir um item (comprar, assistir, etc.). A partir dessa distinção de contexto, os RSs poderiam atribuir pesos diferentes para cada ação, podendo assim fazer recomendações mais precisas.

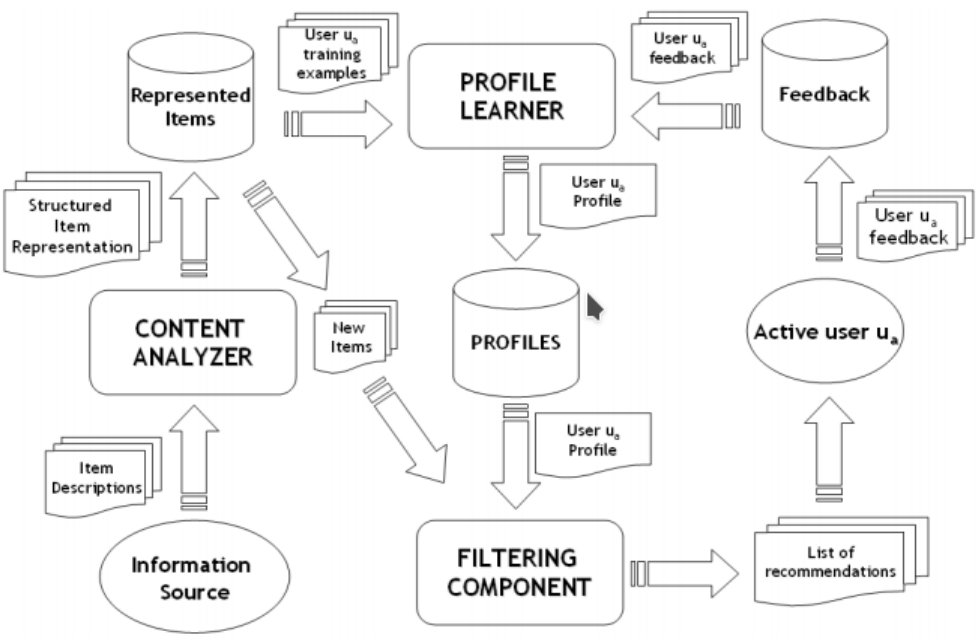
A seguir serão apresentados as diferentes técnicas dos sistemas de recomendação, além de qual técnica será utilizada por este trabalho e seus diferentes métodos através de algoritmos. Devido a existência de inúmeras técnicas e métodos de recomendação, este trabalho abordará apenas as técnicas necessárias para entendimento do mesmo, aprofundando-se apenas nos métodos que compõem a técnica utilizada.

2.1 Método Baseado em Conteúdo

Sistemas de recomendação que implementam o método baseado em conteúdo (*content-based*) analisam um conjunto de documentos/descrições de itens previamente avaliados pelo usuário, construindo um modelo dos interesses baseando-se nas características dos itens avaliados (MLADENIC, 1999; ADOMAVICIUS; TUZHILIN, 2005; LOPS; GEMMIS; SEMERARO, 2011). Este modelo serve para ser cruzado com o conteúdo de outros itens ainda não avaliados pelo usuário. Quanto maior o grau de semelhança entre o modelo do usuário e as características do item, maior a probabilidade do mesmo ter interesse.

Para que o modelo de interesses do usuário seja criado e confrontado com outros conteúdos ainda não avaliados, são necessários três atores principais que dividem a recomendação baseada em conteúdo: **analisador de conteúdo**, **aprendiz de perfis** e **componente de filtragem** (LOPS; GEMMIS; SEMERARO, 2011). A estrutura completa destes agentes pode ser vista na Figura 1.

Figura 1 – Arquitetura de um sistema baseado em conteúdo



Fonte: Lops, Gemmis e Semeraro (2011)

Note que na Figura 1, a primeira parte do processo começa com o **analisador de conteúdo** (*content analyzer*), transformando dados não estruturados em estruturas de atributos e características (LOPS; GEMMIS; SEMERARO, 2011; MLADENIC, 1999), armazenando-as no repositório de itens representados (*represented items*). Para a construção e atualização do perfil de interesses do usuário ativo (representado na Figura 1 por u_a), as avaliações do usuário para novos itens são armazenadas no repositório de feedback. O tipo

de avaliação depende de cada aplicação, podendo ser expressado de forma **explícita**, como as avaliações binárias (*like/dislike*) e avaliações em forma de rating (0 a 5; 1 a 5 estrelas) utilizadas em muitos sites, ou mesmo por avaliações **implícitas**, onde uma ação sobre um item (seleção, por exemplo) possui um peso atribuído (PAZZANI; BILLSUS, 2007).

De posse do repositório de itens representados, o **aprendiz de perfis** varre os itens I_k do usuário ua em prol de construir o conjunto treinamento TR_a. O conjunto de treinamento é um conjunto de pares $\{I_k, r_k\}$, onde r_k é a avaliação dada pelo usuário ua a representação do item I_k. Após a construção do conjunto de treinamento TR_a, o **aprendiz de perfis** aplica algoritmos de aprendizagem supervisionada para gerar o modelo de interesses do usuário u_a. Os modelos de interesses são armazenados no repositório de perfis (representado na Figura 1 por *profiles*) para uso futuro pelo **componente de filtragem**.

Quando a representação de um novo item é adicionada ao conjunto de itens representados, o componente de filtragem prediz se o mesmo será de interesse do usuário u_a, através da comparação entre os atributos e características do novo item e o modelo de interesses do usuário. Em consequência, o componente de filtragem ranqueia os itens com os maiores potenciais de interesse, agrupando-os em uma lista de recomendações L_a e apresentando-a ao usuário u_a. Dessa forma o usuário ua pode prover novas avaliações (**feedback**) dos itens da lista L_a, fazendo com que o aprendiz de perfis atualize seu modelo de interesses através da reconstrução do conjunto de treinamento TR_a (LOPS; GEMMIS; SEMERARO, 2011).

Atualmente Pazzani e Billsus (2007) apresentam que, devido ao grande crescimento de informação disponível para treinamento, os métodos atuais reduzem o conjunto de treinamento para algumas centenas de linhas, porém altamente relevantes (através de técnicas como o TF-IDF [1]). Dessa forma, por mais que as bases de dados aumentem, o conjunto de treinamento se mantém relevante e não é necessário percorrer todo o conjunto ordenado R.

2.2 Método Baseado em Colaboração

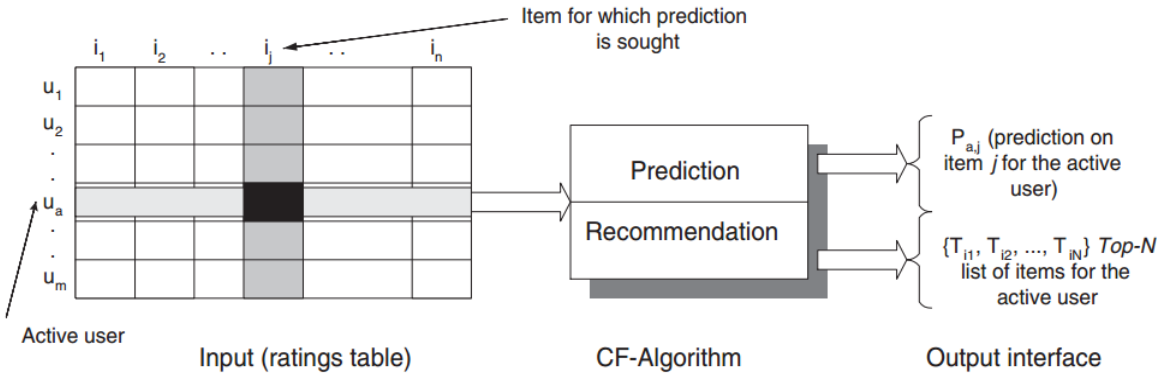
O método de filtragem colaborativa (*collaborative-based* - CF) baseia-se no processo de avaliar itens através da opinião de outras pessoas. Tal processo, que começou com a filtragem da natureza de repositórios de texto, passou a ser mais informal, abrangendo até listas de discussão e arquivos de *e-mail*. No começo, usuários tinham que acessar sites específicos, tais como o MovieLens, para receberem recomendações de filmes. Conforme os sistemas baseados em CF foram se popularizando, os sites começaram a utilizar estes sistemas para adequar seu conteúdo para cada usuário (SCHAFER *et al.*, 2007).

[1] Frequência do termo inverso da frequência nos documentos. Medida estatística para indicar a importância de uma palavra de um documento em relação a uma coleção de documentos. É frequentemente usada na mineração de dados.

Assim como os sistemas baseados em conteúdo, sistemas de filtragem colaborativa também levam em consideração as avaliações de itens (mesmo que de outros usuários similares), através dos métodos de avaliação já descritos. Segundo Adomavicius e Tuzhilin (2005), a diferença entre estes dois processos existe pelo fato de que a utilidade $u(c, s)$ de um item s a um usuário c é medida não pela utilidade $u(c, s_i)$ dos itens s_i similares ao item s, mas sim pela utilidade $u(c_j, s)$ do item s baseado nos usuários c_j **similares** ao usuário c. Em outras palavras, na filtragem colaborativa, os itens considerados úteis a um usuário são os itens úteis a usuários similares a ele.

Partindo desta premissa, Sarwar *et al.* (2001) abordam os sistemas de filtragem colaborativa a partir do seguinte cenário: uma lista de **m** usuários $U\{u_1, u_2, \ldots, u_m\}$ e uma lista de **n** itens $I\{i_1, i_2, \ldots, i_n\}$. Cada usuário u_i possuindo uma lista Iu_i de itens, avaliados ou não. Conforme na Figura 2, o algoritmo de filtragem colaborativa (**CF**) opera sobre a matriz de avaliações $n \times m$.

Figura 2 – Processo de recomendação colaborativa.



Fonte: Sarwar *et al.* (2001)

De posse da matriz $n \times m$, o algoritmo **CF** faz a predição/recomendação ao usuário corrente, demonstrado na Figura 2 por u_a. O usuário u_a é visto pelo algoritmo como o alvo atual para o qual serão feitas as predições/recomendações. Sarwar *et al.* (2001) também especificam a predição como um valor numérico que expressa a probabilidade prevista do item ser de interesse do usuário u_a, sendo este um item ainda não pertencente ao conjunto de Iu_a. Por outro lado, a recomendação é descrita como uma lista de **N** itens, cada item I_r dentre os itens com a maior probabilidade de utilidade ao usuário ua e ainda não avaliados pelo mesmo. Esta forma de recomendação também é conhecida como recomendação **Top-N** (ADOMAVICIUS; TUZHILIN, 2005).

Diferentemente do método baseado em conteúdo, a filtragem colaborativa não possui apenas uma abordagem. Tanto Sarwar *et al.* (2001 apud BREESE; HECKERMAN; KADIE, 1998) quanto Adomavicius e Tuzhilin (2005) dividem a filtragem colaborativa em

duas ramificações:

- **Baseada em memória (*memory-based*)**: implica na utilização de toda a matriz $n \times m$ para obter um conjunto de usuários vizinhos (*neighbor-users*), ou seja, usuários que tendem a avaliar diferentes itens similarmente ou itens similares diferentemente ao usuário u_a. Ao obter o conjunto, os métodos baseados em memória combinam as preferências dos usuários, fornecendo uma recomendação Top-N ao usuário u_a.
- **Baseada em modelo (*model-based*)**: ao invés de utilizar toda a matriz $n \times m$, esta técnica constrói um modelo das avaliações de cada usuário através de diferentes técnicas de *machine learning*, tais como modelos de *cluster* e redes Bayesianas. Devido a complexidade destas técnicas e das mesmas não pertencerem ao escopo da solução apresentada neste trabalho , não abordaremos mais a fundo seu funcionamento.

Dessa forma, sistemas de filtragem colaborativa podem ser usados nos casos em que se deseja recomendar itens úteis a um usuário ou fornecer uma previsão ao usuário da probabilidade do mesmo gostar de um item em particular. Além disso, é possível recomendar ao usuário não só itens, mas também usuários ou grupos de usuários que o mesmo possa gostar, o que não é possível nos sistemas baseados em conteúdo (SCHAFER *et al.*, 2007).

Considerando tais utilidades, tanto Schafer *et al.* (2007) quanto Adomavicius e Tuzhilin (2005) expõem os sistemas de recomendação baseados em conteúdo e colaborativos como complementares, uma vez que o método baseado em conteúdo prediz a relevância de itens sem avaliações, enquanto o método colaborativo prediz a relevância através de recomendações alheias. A união destas técnicas, em prol de maximizar a eficiência e compensar as limitações (seção 2.2.4), deu origem ao **método híbrido** que será abordado a seguir.

2.3 Método Híbrido

Sistemas de recomendação híbridos seriam quaisquer sistemas que combinam múltiplas técnicas de recomendação para produzir seu resultado (BURKE, 2002; BURKE, 2007). Como apresentado por Adomavicius e Tuzhilin (2005), as técnicas de recomendação possuem limitações de acordo com a abordagem utilizada. Sendo assim, é possível combinar diferentes técnicas para obter o desempenho e precisão desejadas.

Como pode ser visto no quadro 2, Burke (2002) apresenta uma série de métodos de recomendação além dos mais comuns abordados neste trabalho. Estes métodos, combinados entre si, podem gerar sistemas híbridos categorizados da seguinte forma:

Quadro 2 – Técnicas de recomendação.

Technique	Background	Input	Process
Collaborative	Ratings from **U** of items in **I**.	Ratings from **u** of items in **I**.	Identify users in **U** similar to **u**, and extrapolate from their ratings of **i**.
Content-based	Features of items in **I**	**u**'s ratings of items in **I**	Generate a classifier that fits **u**'s rating behavior and use it on **i**.
Demographic	Demographic information about **U** and their ratings of items in **I**.	Demographic information about **u**.	Identify users that are demographically similar to **u**, and extrapolate from their ratings of **i**.
Utility-based	Features of items in **I**.	A utility function over items in **I** that describes **u**'s preferences.	Apply the function to the items and determine **i**'s rank.
Knowledge-based	Features of items in **I**. Knowledge of how these items meet a user's needs.	A description of **u**'s needs or interests.	Infer a match between **i** and **u**'s need.

Fonte: Burke (2002)

- **Atribuição de peso (*Weighted*):** Consiste na atribuição de peso para cada um dos métodos empregados no sistema híbrido. Baseado no histórico de acertos entre um método e outro, é possível ajustar o peso de cada um, dando um peso maior ao método atualmente mais eficiente.

- **Escalonamento (*Switching*):** Consiste na utilização de um critério pré-definido para escolher qual método será utilizado no momento. Por exemplo, se o método colaborativo não fornecer uma recomendação com confiança suficiente, o sistema pode trocar para o método baseado em conteúdo.

- **Misto (*Mixed*):** Consiste em usar tanto recomendações de um método quanto de outro, apresentando os resultados de ambos ao usuário.

- **Combinação de características (*Feature Combination*):** Consiste em utilizar a informação colaborativa apenas como características adicionais no conjunto utilizado pelo método baseado em conteúdo.

- **Cascata (*Cascade*):** Este método em especial consiste em refinamento por estágio, ou seja, o primeiro método é utilizado para gerar um conjunto de recomendações, enquanto o segundo é responsável por refinar o conjunto gerado e assim por diante.

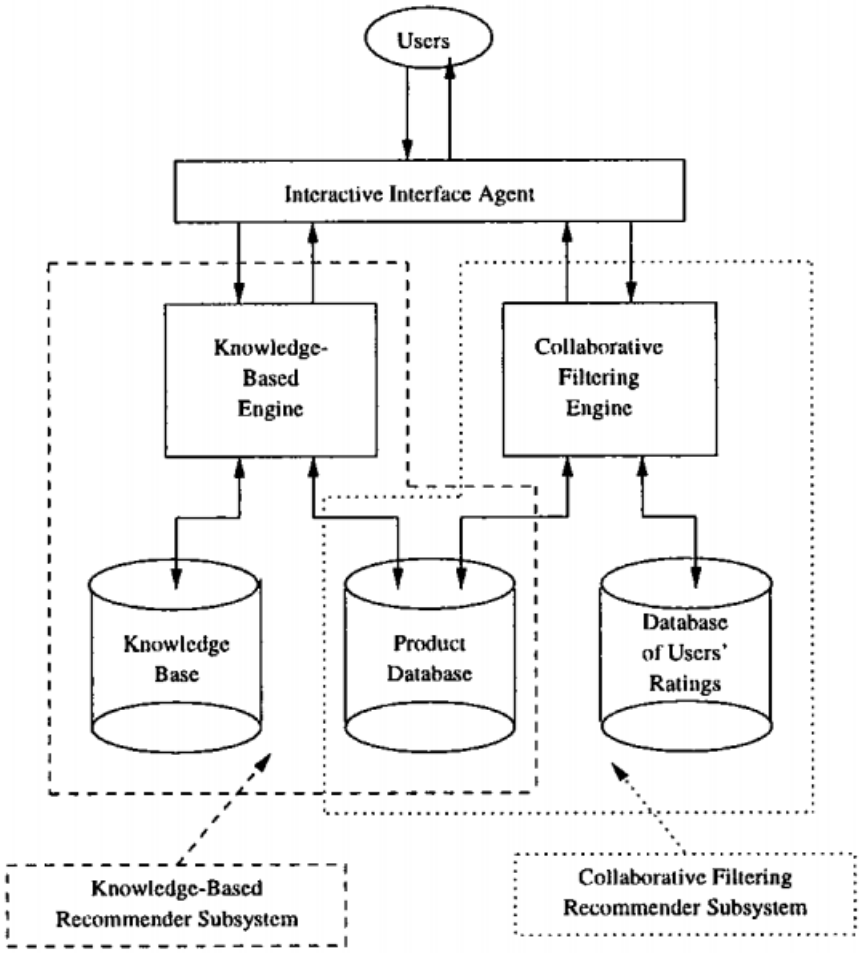
- **Aumento de Recursos (*Feature Augmentation*):** Esta técnica utiliza a recomendação gerada pelo primeiro método como informação para o processamento do segundo método.

- **Meta-nível (*Meta-level*):** Consiste em utilizar o modelo de saída de um método como entrada para o outro. Diferente do aumento de recursos, nesta técnica todo o modelo gerado pelo primeiro método é utilizado.

Tendo em vista a taxonomia apresentada por Burke (2002), nota-se que a recomendação híbrida não refere-se ao funcionamento das recomendações, mas sim sobre como os diferentes métodos **interagem entre si**. Esta interação pode ser insensível à ordem, nos casos de métodos como a atribuição de peso, misto, escalonamento e combinação de características. Já nas outros métodos apresentados, a ordem de execução dos métodos de recomendação alteram o resultado final, uma vez que a saída de um, direta ou indiretamente é a entrada de outro.

Por exemplo, Tran e Cohen (2000) apresentam uma arquitetura híbrida, utilizando os métodos baseado em colaboração (collaborative-based) e baseado em conhecimento (knowledge-based), ambos exemplificados através da arquitetura ilustrada na Figura 3.

Figura 3 – Exemplo de arquitetura híbrida.



Fonte: Tran e Cohen (2000)

Como ilustrado na Figura 3, a arquitetura descrita exemplifica um sistema híbrido de escalonamento (*switching*). Sendo assim, dependendo da situação atual, o sistema pode trocar entre a recomendação colaborativa e a baseada em conhecimento, visando prover melhores recomendações. Considerando que inicialmente a abordagem colaborativa não seria muito eficiente, enquanto a base de dados não possui muitos usuários com modelos conhecidos e não existem itens avaliados o suficiente, Tran e Cohen (2000) optaram por escalonar para o método baseado em conhecimento.

Através dessas limiares, toda vez que o usuário requisita uma recomendação, o agente de interface interativa (*interactive interface agent*) verifica se as mesmas já foram atendidas. Se sim, o agente utiliza a recomendação do método de filtragem colaborativa, se não, o método baseado em conhecimento é utilizado.

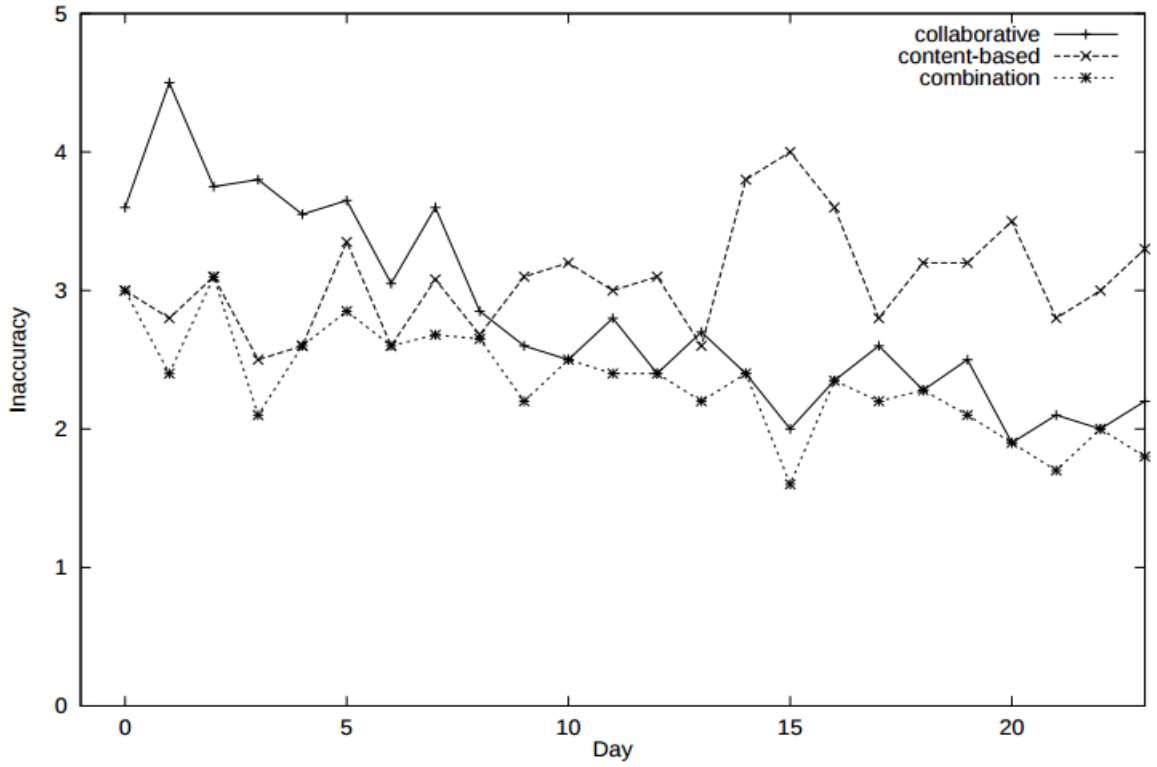
Tanto Balabanović e Shoham (1997) quanto Claypool *et al.* (1999) utilizam sistemas híbridos compostos de duas técnicas combinadas: **baseado em conteúdo** e **baseado em colaboração**. Dessa forma, é possível utilizar o método colaborativo para gerar o conjunto de **N** usuários vizinhos (*neighbor-users*) já descrita neste trabalho. A partir do conjunto gerado é aplicado o método baseado no conteúdo destes usuários próximos, aumentando a precisão da recomendação gerada.

Ao invés de se utilizar apenas um método, a utilização de sistemas híbridos pode trazer uma série de benefícios: ao executar recomendações baseadas em conteúdo, o sistema colaborativo pode lidar com novos usuários que ainda não tem seu modelo definido; torna-se possível fazer recomendações precisas a um usuário, mesmo que não existam usuários similares ao mesmo; pode-se recomendar itens não avaliados por nenhum dos usuários, cruzando o modelo dos mesmos com o conteúdo do item (BALABANOVIĆ; SHOHAM, 1997).

Como forma de verificar a eficácia do método híbrido em relação aos métodos utilizados de forma individual, Claypool *et al.* (1999) utilizam como métrica a inexatidão, sendo o termo referente a discrepância entre a recomendação obtida e o resultado esperado. A inexatidão dos métodos em relação a seu tempo de utilização pode ser visto através do resultado ilustrado na Figura 4.

Analisando a Figura 4 pode-se verificar que nos primeiros dias, a inexatidão do método colaborativo era maior devido a falta de completude no modelo dos usuários, construído por meio de usuários que ainda não avaliaram itens, ou de usuários que não se beneficiam da opinião de outros (CLAYPOOL *et al.*, 1999). Conforme o método colaborativo foi estabelecendo relações entre os usuários, este ficou mais preciso e o método baseado em conteúdo começou a ser menos viável. Porém, independente dos picos de inexatidão dos métodos separados mostrados na Figura 4, quando combinados (**recomendação híbrida**), é possível notar uma constância muito maior, possuindo o mais baixo nível de inexatidão em todos os momentos.

Figura 4 – Inexatidão entre os métodos de recomendação.



Fonte: Claypool *et al.* (1999)

Em resumo os sistemas híbridos foram criados para unir técnicas de recomendação com objetivo de **compensar as limitações** apresentadas pela utilização dessas mesmas técnicas individualmente (BALABANOVIĆ; SHOHAM, 1997). Tais limitações e seus efeitos no resultado das recomendações serão abordadas na seção a seguir.

2.4 Limitações

Conforme apresentado por Adomavicius e Tuzhilin (2005), decorrente da utilização das técnicas acima descritas, tanto os sistemas baseados em conteúdo quanto os sistemas colaborativos possuem limitações. Estas limitações, motivo da criação dos sistemas híbridos (BALABANOVIĆ; SHOHAM, 1997), possuem características claras de acordo com o tipo de recomendação utilizado, sendo divididas da seguinte maneira:

- **Análise de conteúdo limitada (*limited content analysis*):** presente nas técnicas baseadas em conteúdo, devido as mesmas serem limitadas por uma quantidade específica de características relevantes para a recomendação. Além disso, essas características precisam ser extraídas de forma explícita, o que dificulta muito a extração de atributos através de conteúdo como vídeo, imagem, etc.

- **Problema do novo usuário (*new user problem*):** comum nas técnicas que utilizam as preferências do usuário como métrica, consiste no fato de que um usuário precisa ter um número suficiente de avaliações para que o sistema entenda suas preferências e forneça recomendações precisas.

- **Superespecialização (*over-specialization*):** comum nas técnicas de recomendação baseada em conteúdo, consiste no fato de que se o sistema apenas recomenda ao usuário itens semelhantes aos que ele já avaliou de forma positiva, o usuário será limitado à apenas recomendações de itens já avaliados, reduzindo cada vez mais a recomendação de novos itens.

- **Problema do novo item (*new item problem*):** sistemas colaborativos baseiam-se apenas nas preferências dos usuários para fazer as recomendações. Sendo assim, novos itens que ainda não foram avaliados por nenhum usuário não serão recomendados.

- **Esparsidade (*Sparsity*):** quando um item é raramente recomendado devido a sua esparsidade no conjunto de usuários e itens, ou seja, um item que é pouco recomendado pelos usuários tende a ser cada vez menos recomendado em sistemas colaborativos, devido ao pouco número de avaliações que o mesmo possui.

Sendo assim, grande parte das pesquisas relacionadas a sistemas de recomendação tem como objetivo principal melhorar a precisão das técnicas, reduzindo o impacto das limitações descritas. Porém, como apresentado por McNee, Riedl e Konstan (2006), nem sempre os itens mais precisos em relação às métricas de cada método são os mais úteis aos usuários.

Considerando que os sistemas de recomendação usualmente abordam apenas algumas, das muitas métricas que definem a utilidade de um item ao usuário, McNee, Riedl e Konstan (2006) ressaltam que cada vez mais a utilização de sistemas de recomendação leva a construção de um conjunto de itens muito similar. Isso ocorre pois quando um usuário avalia um item, as próximas recomendações levarão o mesmo em consideração, recomendando itens cada vez mais parecidos com o item avaliado. Este processo acaba gerando o que McNee, Riedl e Konstan (2006) definem como "**buraco de similaridade**", onde o sistema tende a fazer apenas recomendações excepcionalmente similares.

3 API: *Application Programming Interface*

Seção sobre API

3.1 O Padrão RESTful

Descrever o padrão RESTful

3.2 Orientação à Metadados

Encaixar os metadados aqui

3 API DE RECOMENDAÇÃO ORIENTADA À METADADOS

Amparado pelos benefícios da utilização de mais de um método de recomendação, apresentando-os em forma de serviço, este capítulo detalha o funcionamento de uma **API como interface de recomendação**, fazendo uso de metadados provenientes do próprio usuário para criação, validação e interconexão entre as recomendações.

Como parte introdutória ao funcionamento, a seção Visão Geral traça de forma abstrata as funcionalidades da API, apresentando os principais componentes e sua correlação. Posteriormente, nas seções O Analisador de Dados, O Motor de Recomendações e A Interface de Comunicação, tais funcionalidades são dissecadas de forma diminuir o nível de abstração. Por fim, os testes efetuados e as facilidades implementadas para a utilização da solução são abordados na seção Testes e Implementação.

1 Visão Geral

A solução construída tem como função recomendar quaisquer itens a quaisquer usuários, sendo estes provenientes de uma fonte externa, fornecidos pelo usuário da API, nos padrões definidos pelos respectivos **metadados** existentes, também fornecidos inicialmente pelo usuário da API.

O usuário da API é definido como qualquer agente que tenha interações com a aplicação, humano ou programa, através requisições aos *endpoints* fornecidos na documentação. Essas interações são acordadas pelo protocolo HTTP, usado como base em uma arquitetura RESTful (RODRIGUEZ, 2008). Dessa forma, o usuário da API pode utilizar qualquer interface que suporte o protocolo HTTP para desfrutar das funcionalidades da API, tornando-a multiplataforma e independente de bibliotecas de linguagens de programação específicas. As interações com os recursos da aplicação são efetuadas através dos seguintes métodos do protocolo HTTP:

- **GET**: Interações que demandam a consulta de recursos.
- **POST**: Interações que demandam a criação de novos recursos.
- **PUT**: Interações que demandam a alteração de recursos existentes.
- **DELETE**: Interações que demandam a remoção ou desativação de recursos existentes.

Após o envio, a requisição é processada pela interface de usuário da API e, posteriormente, um retorno de sucesso ou erro finaliza a requisição. Para verificar o tipo do retorno da requisição, o usuário da API deve se ater aos *status code* retornados pela API. Em caso de sucesso, por exemplo, a API retornaria um conteúdo JSON relacionado a requisição original e um *status code* da faixa **2XX**. Já em caso de erro, a API retornaria um conteúdo de erro, anexo a um *status code* da faixa **4XX** (FIELDING *et al.*, 1999). Exemplos de requisição à API e o fluxo das mesmas após o envio podem ser visualizados nas figuras 5, 6 e 7.

Figura 5 – Exemplo de adição de item via *fetch* API.

```
fetch('morpyapi.com/item', {
    method: 'post',
    headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'token': 'eyJhbGciOiJIUzI1NiIsImV4cCI6MTUwODE4OTI0MywiaWF0IjoxNTA2ODkzMjQzfQ'
    },
    body: JSON.stringify({
        'title': 'The Mummy (1999)',
        'director': 'Stephen Sommers',
        'duration': 124
    })
}).then(response => response.json())
  .then(json_response => console.log(response))
  .catch(error => console.log(error));
```

Fonte: O Autor.

Figura 6 – Exemplo de adição de item via Postman.

HTTP Copy to Clipboard

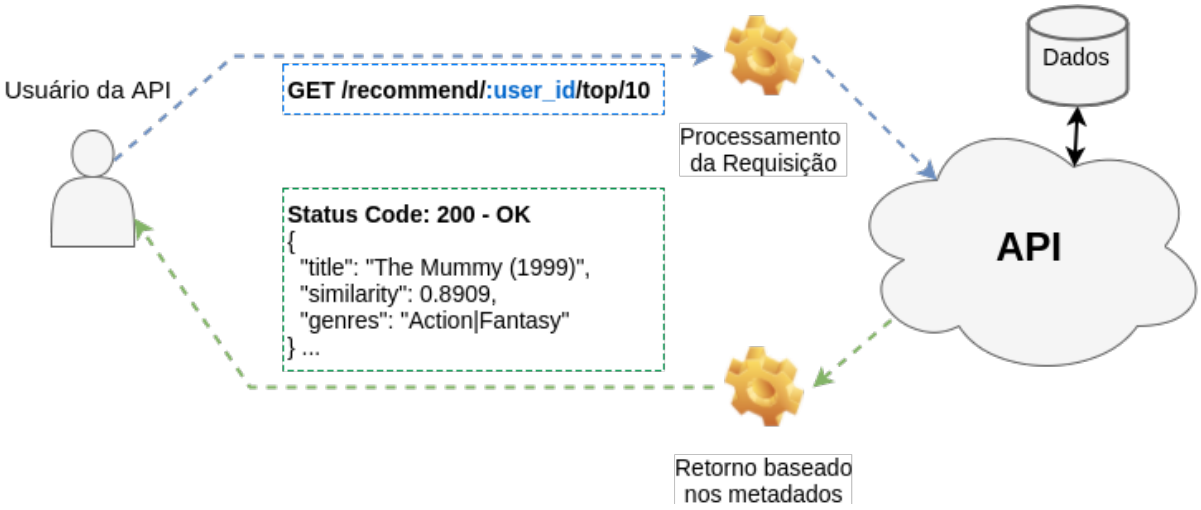
```
POST /item HTTP/1.1
Host: 127.0.0.1:5000
Content-Type: application/json
token: eyJhbGciOiJIUzI1NiIsImV4cCI6MTUwODE4OTI0MywiaWF0IjoxNTA2ODkzMjQzfQ
    .eyJwYXNzd29yZCI6IjEyMyIsImVtYWlsIjoiaXVyeS5rcmllZ2VyQGNoYW9yZGljc3lzdGVtcy5jb20ifQ
    .FAydYiQRJowfjaGmjJX0ZdLN1AI8PUigN5B4gJvy9Dk
Cache-Control: no-cache
Postman-Token: 4d02f3d4-4c87-4416-11d4-fdb8792ec1e1

{
    "genres": "Action|Adventure",
    "title": "The Mummy (1999)",
    "description": "An American serving in the French Foreign Legion on an archaeological dig at the
        ancient city of Hamunaptra accidentally awakens a mummy that wreaks havoc on him, and his crew
        ."
}
```

Fonte: O Autor.

A figura 7 mostra um exemplo de requisição à API de recomendação. Nela, são requisitados os **dez** itens de maior similaridade a serem recomendados para o usuário denominado pelo parâmetro variável ***user_id***. Este processo de requisição e resposta pode ser dividido em quatro partes principais:

Figura 7 – Exemplo de fluxo de uma requisição à API.



Fonte: O Autor.

1. **Envio**: O usuário da API, devidamente autenticado, escolhe o *endpoint* que corresponde as suas necessidades, enviando uma requisição através dos métodos HTTP previamente citados.

2. **Interpretação**: A API receberá a requisição no devido *endpoint*, interpretando-a e repassando-a através de chamadas internas das devidas funcionalidades.

3. **Processamento**: Uma vez identificada a ação a ser executada, a API processa os dados (nesse caso as recomendações para o usuário ***user_id***), posteriormente formatando-os em uma estrutura JSON.

4. **Retorno personalizado**: Com a estrutura JSON processada em mãos, a API verifica os metadados atuais e retorna, com base nos atributos especificados como visíveis, uma estrutura JSON personalizada pelos desejo do usuário da API.

Partindo da premissa que a solução deve atender modelos genéricos, serão fornecidos na inicialização da API os **metadados** de usuários, itens e avaliações, correspondendo a estrutura necessária pelo usuário da aplicação. Uma vez que os metadados sejam fornecidos, os dados relacionados devem respeitar as estruturas definidas. Tanto os usuários, itens e avaliações, quanto as futuras recomendações, serão persistidas em um banco de dados, a fim de centralizar as informações e diminuir o tempo de resposta das recomendações requisitadas.

De posse das estruturas de metadados fornecidas na inicialização, o usuário da solução poderá alimentar o sistema através das seguintes interfaces gerais:

- **Metadados**: O usuário da API fornece a estrutura que irá compor cada um dos grupos abaixo. Essas estruturas definem os atributos de cada grupo, o tipo de cada atributo, a importância ou não do atributo nas recomendações, entre outras informações.

- **Usuários**: O usuário da API fornece os usuários aos quais deseja gerar algum tipo de recomendação. Esses dados serão utilizados posteriormente como referência aos itens e para definição da similaridade entre usuários.

- **Itens**: O usuário da API fornece os itens que serão recomendados aos usuários existentes. Os mesmos e seus atributos serão utilizados nas recomendações, atrelados a um usuário em questão.

- **Avaliações**: O usuário da API fornece as avaliações que ligam usuários a itens. Essas informações servem como união, apontando quais usuários estão relacionados a quais itens, sempre atrelando esta relação a uma avaliação numérica.

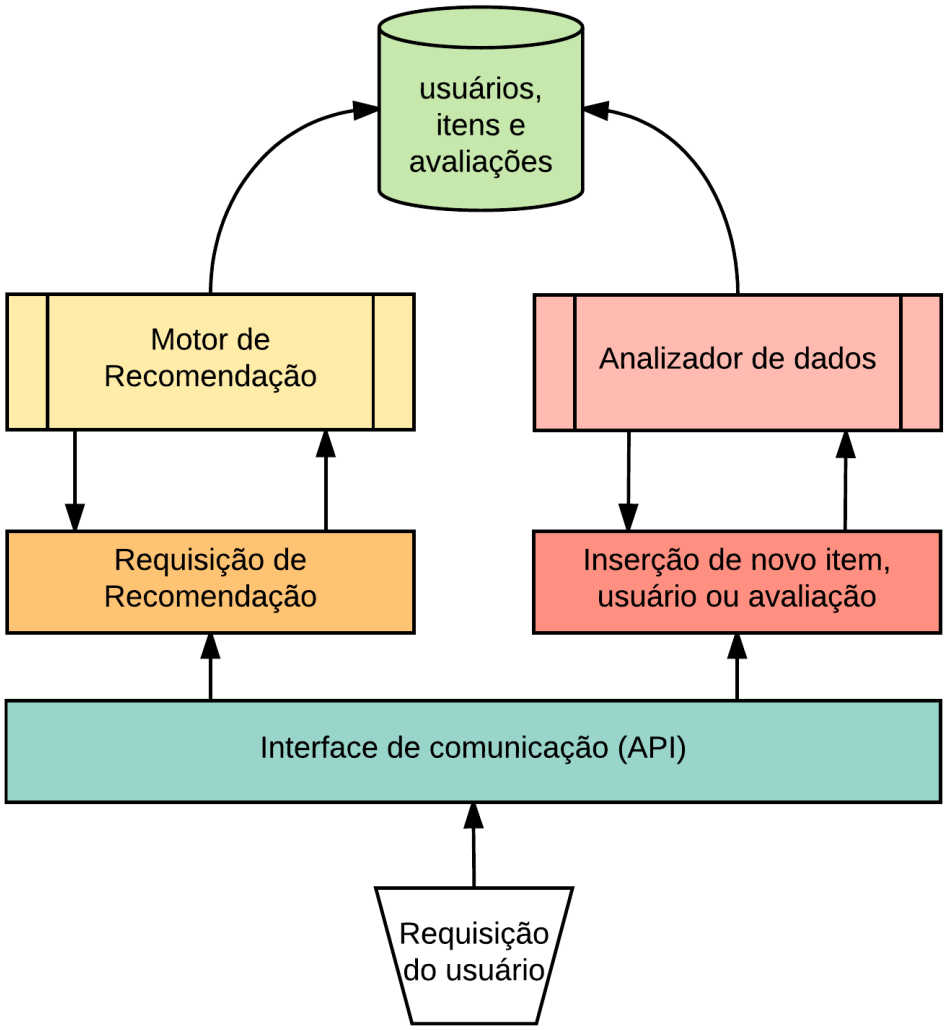
Tais ações de alimentação serão responsáveis por preencher os respectivos conjuntos anteriormente descritos e, a partir deles, construir os modelos de cada usuário e a matriz de avaliações, fundamentais para a geração das recomendações. Durante a requisição de uma recomendação, a solução definirá o método a ser utilizado e o mesmo fará as recomendações com base nos dados conhecidos, respeitando as propriedades descritas nos metadados. Um esquema do funcionamento geral da aplicação pode ser visto na Figura 8.

De modo a complementar a visão da API apresentada pela figura 7, a figura 8 apresenta o fluxo interno de funcionamento da API, apresentando uma distinção básica entre dois tipos de eventos: os eventos de **inserção ou atualização de dados** e os eventos de **geração e requisição de recomendações**. Ambos estão contidos no domínio de atuação da interface de comunicação com o usuário, a qual interpreta as requisições e processa as devidas respostas. Os módulos centrais correspondentes a estes domínios e suas interações podem ser subdivididas nos seguintes:

- **Interface de Comunicação**: É o ponto de contato entre o cliente que utiliza a API e o domínio da aplicação, recebendo todas as requisições nos devidos *endpoints*, processando-as e retornando os resultados. Este módulo é descrito detalhadamente na seção A Interface de Comunicação.

- **Analisador de Dados**: É o módulo responsável por contrastar as entradas da interface de comunicação com os metadados ativos. O trabalho do analisador de dados é validar o padrão de dados fornecido pelo cliente da API e persistir os dados corretos na base dados, notificando o motor de recomendação sobre quaisquer mudanças. Este módulo é descrito detalhadamente na seção O Analisador de Dados.

Figura 8 – Visão geral da API.



Fonte: O Autor.

- **Motor de Recomendação**: É o módulo que definitivamente faz as recomendações. Ele se alimenta dos dados persistidos pelo analisador, gerando matrizes de relação, utilizadas para calcular o nível de similaridade entre usuário e itens, permitindo a persistência das recomendações na base de dados. Este módulo é descrito detalhadamente na seção O Motor de Recomendações.

Assim que uma requisição é recebida pela interface de comunicação, se a mesma for uma requisição de recomendação, a interface delega ao **motor de recomendações**, que busca as recomendações persistidas para o usuário em questão. Caso a requisição seja uma alteração de dados existentes, a interface delega ao **analisador de dados** a tarefa de validar, sempre baseando-se nos metadados atuais, os novos dados providos pelo usuário da API. Assim que esses dados são persistidos na base, o **motor de recomendações**

identifica uma mudança na base e inicia um novo processo paralelo de treinamento para o dado modificado. Dessa forma, quando uma requisição for solicitada, basta o motor de recomendações consultar as recomendações já persistidas.

2 A Interface de Comunicação

Visando facilitar a comunicação com o usuário da API através do padrão RESTful, a interface de comunicação tem como principal função ser o ponto de contato entre o agente da requisição e as funcionalidades da API, atendendo a certos padrões de requisição e resposta que serão minuciosamente abordados nesta seção.

Ao passo que a API usa o padrão RESTful como base para a comunicação, é necessário que a mesma utilize um único padrão de dados para que ambos, usuário e API, tenham uma estrutura pré definida de possibilidades de envio e retorno. Dessa forma, o padrão de dados **JSON**, proposto por Crockford (2006) foi o escolhido para servir como representação de dados da API, desde o recebimento de novos recursos (itens, usuários, etc.) até o retorno das recomendações. Um exemplo da representação JSON para as recomendações pode ser visto na figura 9.

Figura 9 – Exemplo de estrutura JSON para recomendações.


```
[
  {
        "similarity": 0.2709252651331338,
        "genres": "Crime|Drama",
        "_id": "WdY_X9w2E1X88m9J",
        "title": "Twilight (1998)"
    },
    {
        "similarity": 0.2709252651331338,
        "genres": "Crime|Drama",
        "_id": "WdY_X9w2E1X88nDY",
        "title": "Phoenix (1998)"
    }, //...
]
```


Fonte: O Autor.

Em outras palavras, esta seção apresenta o ponto de contato com a API, onde todas as requisições para modificações de recursos são feitas, visando melhorar a precisão

das recomendações. Primeiro, é abordado o padrão de nomenclatura dos *endpoints* na seção 2.1, de modo a esclarecer de forma geral, toda a gama de possibilidade que o usuário da API tem ao utilizá-la. Em seguida, é apresentado o processo de autenticação na seção 2.2, necessário para que o usuário da API possa executar as requisições.

2.1 Padrão de *Endpoints*

Toda a comunicação entre o agente das requisições e a API funciona através de *endpoints* que modificam seus respectivos recursos. Recursos esses que são abstrações da base de dados, fazendo com que o agente das requisições possa interferir diretamente na evolução das recomendações, mesmo sem que haja uma interface gráfica. Um exemplo de recurso pode ser visualizado no quadro 3.

Quadro 3 – Exemplo de recurso de *endpoints* para o módulo item.

Método HTTP	***Endpoint***	**Descrição**
POST	/item	Cria um novo item
GET	/item/:item_id	Retorna um item específico
PUT	/item/:item_id	Altera um item específico
DELETE	/item/:item_id	Deleta um item específico
GET	/item	Retorna todos os itens

Fonte: O Autor.

Analisando o quadro 3, nota-se que um mesmo *endpoint* pode apontar para diferentes funções, uma vez que o método HTTP que encabeça a requisição também é levado em consideração. Além disso, é importante ressaltar que cada par de **método HTTP** e ***endpoint*** correspondem diretamente a um **controlador** da API, que recebe os parâmetros variáveis da requisição (no quadro 3 representando por *item_id*) e os repassa aos serviços. Sendo assim, os controladores poderiam ser definidos como as "portas de entrada"da API, direcionando a execução de cada funcionalidade.

Para aumentar o nível de abstração do funcionamento da API, o padrão de *endpoints* utiliza do atributo de identificador único de cada módulo, previamente definido nos metadados de itens, usuários e avaliações, para o acesso a um recurso específico. Dessa forma, o usuário da API pode utilizar os seus próprios identificadores de sua base de dados externa, para acessar os mesmos recursos na base interna da aplicação. O modo de declaração deste atributo e seu tipo de dado esperado podem ser vistos na seção Padrão de Metadados.

[TODO] Fazer uma tabela com todos os endpoints.

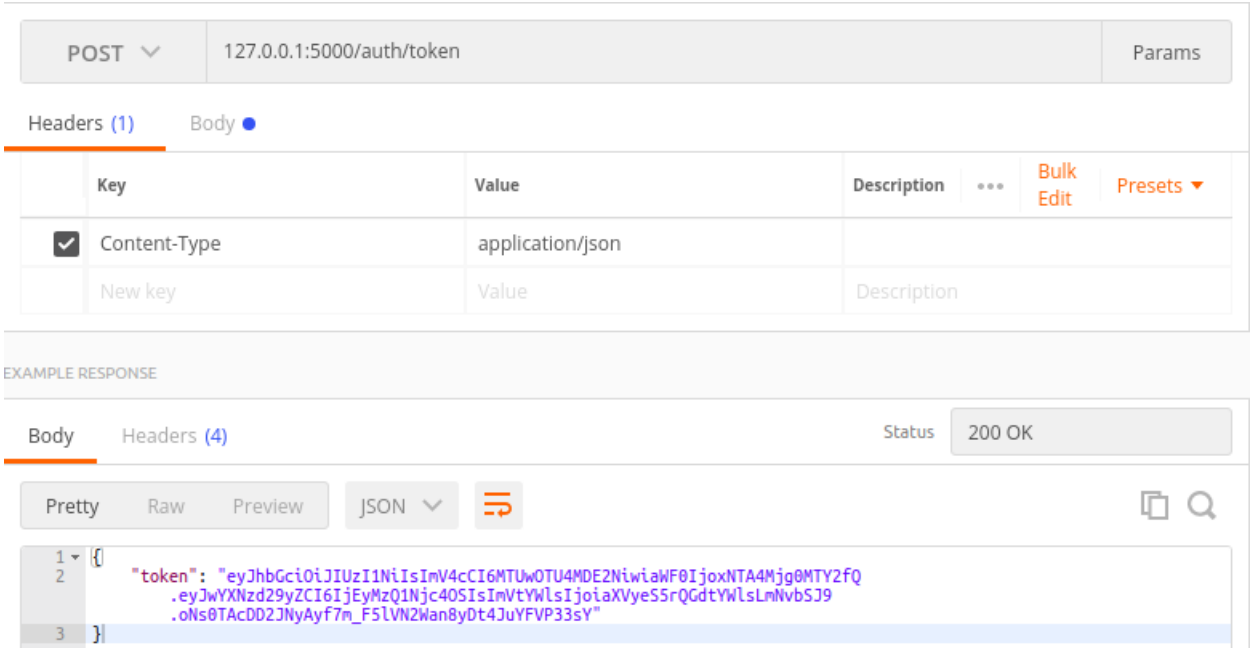
2.2 Autenticação

Como forma de aumentar a segurança durante as requisições e, principalmente, garantir que apenas os usuários com acesso ao serviço façam requisições, a API conta com um sistema de autenticação via *token*.

Antes da primeira inicialização, o *script* gera um *token* único para o usuário da API, que será posteriormente concatenado com uma palavra secreta nos arquivos de configuração, além da data e hora atuais da requisição de autenticação. O resultado final é um *token* temporário, com validade de até quinze dias, que será utilizado, obrigatoriamente, no cabeçalho de quaisquer requisições aos recursos da API.

Do mesmo modo que o agente das requisições precisa incluir o *token* temporário no cabeçalho das requisições, a API precisa verificar a validade deste *token*. Para que isso seja possível, foi criada uma classe intermediária (*middleware*) que intercepta todas as requisições, exceto a requisição de autenticação. Essa classe, por sua vez, verifica a existência do *token* no cabeçalho da requisição bem como sua validade. Caso o *token* não exista ou não remeta a um usuário de API válido, o *middleware* bloqueia a execução da requisição, retornando uma mensagem de erro e um *status code* igual a **403** - "*Forbidden*". Um exemplo requisição de autenticação pode ser visto na figura 10.

Figura 10 – Exemplo de autenticação.



Fonte: O Autor.

3 O Analisador de Dados

Para tornar possível a geração de recomendações de forma genérica, através dos metadados, foi necessária a construção de um módulo dedicado exclusivamente a este fator. O **analisador de dados** tem como principal objetivo cruzar as novas informações, repassadas pela interface de comunicação, com os metadados correntes.

Por exemplo, para que um novo item seja adicionado ao conjunto de itens conhecidos e, posteriormente, treinado contra os outros itens, é necessário que o analisador de dados verifique se todos os atributos relevantes para as recomendações estão preenchidos, se os tipos dos campos fornecidos condizem com os tipos especificados nos metadados, etc.

Em síntese, esta seção apresentará os principais componentes do analisador de dados. Primeiro apresentando os padrões utilizados para a orientação a metadados na seção 3.1, em seguida, os modelos gerados pelos metadados e sua utilização em toda a aplicação, aprofundados na seção 3.2 e, por fim, como é feita a persistência destes dados na seção 3.3.

3.1 Padrão de Metadados

Uma vez que não se sabe qual o padrão de JSON a se retornar ao usuário, ou quais os campos relevantes para a geração das recomendações, ou ainda quais sequer são os nomes dos atributos, os metadados são as estruturas, previamente definidas e passíveis de personalização, fornecidas pelo usuário na inicialização da aplicação. Estas estruturas são enviadas através de uma estrutura JSON para os *endpoints* respectivos, divididos em três categorias:

- **Usuários**: Definem os nomes dos atributos, obrigatórios ou não, que compõem o modelo de usuário, os tipos de cada atributo, a quantidade máxima de caracteres, se é recomendável ou não e seu peso. Além disso os metadados de usuários apontam o identificador único do usuário.
- **Itens**: Assim como os usuários, definem os nomes de cada atributo, obrigatórios ou não, do item e suas demais características. Além disso, é definida a exibição de cada atributo do item no retorno das recomendações e seu identificador único.
- **Avaliações**: Definem obrigatoriamente o campo de identificador único do usuário e o campo de identificador único do item, amarrando efetivamente as duas entidades. Além disso, definem o tipo de avaliação que será empregada (binária ou não).

Para ilustrar o processo de definição dos metadados, a figura 11 mostra um exemplo real dos metadados inciais para o modelo de **item**. Estes atributos podem ser modificados

posteriormente através do mesmo *endpoint* utilizando o método PUT do HTTP. Caso o qualquer atributo sofra posterior alteração, os metadados são salvos com o status "***active:false***", orientado que o mesmo não é mais válido. Em seguida, um novo conjunto de metadados é criado com o atributo "***active:true***" e as modificações solicitadas pelo usuário.

Figura 11 – Exemplo de declaração de metadados para o item.

```
{
    "type" : "item",
    "attributes": [
        {
            "name": "id",
            "key": true,
            "type": "integer",
            "recommendable": false,
            "hide": true
        },
        {
            "name": "title",
            "weight" : 10,
            "unique": true,
            "type" : "string",
            "max_length": 120,
            "recommendable": true
        },
        {
            "name": "genres",
            "weight": 10,
            "unique": false,
            "type": "string",
            "max_length": 360,
            "recommendable": true
        },
        {
            "name": "description",
            "weight" : 10,
            "unique": false,
            "type": "string",
            "recommendable": true
        }
    ]
}
```

Fonte: O Autor.

Analisando a figura 11 nota-se um conjunto de chaves pré-definidas para a estrutura declarada. A razão dessas chaves possuírem suas definições **obrigatórias e pré-definidas** é o fato de que as mesmas moldam todas as outros atributos dinamicamente atribuídos aos modelos de itens, usuários e avaliações, consequentemente moldando o banco de dados. Sendo assim, cada chave da estrutura definida na figura é utilizada como atributo chave para alguma decisão estrutural dentro da API. As chaves definidas na figura possuem as

seguintes funcionalidades:

- ***type***: Fora da chave ***"attributes"***, define o tipo do grupo de metadados que correspondem os atributos declarados. Dentro de um atributo, define o tipo do dado ao qual o atributo espera um valor, podendo ser qualquer tipo natural: ***string***, ***integer***, ***float***, etc. Espera um valor do tipo ***string***.
- ***attributes***: Declara a lista de atributos pertencentes ao *type*, neste caso ao item. Espera um valor do tipo ***array***.
- ***name***: Define o nome de um atributo, o qual será utilizado como chave do atributo no banco de dados e na construção dinâmica do modelo. Espera um valor do tipo ***string***.
- ***key***: Define se o atributo é ou não um identificador único. Será utilizado posteriormente para ligar os usuários a itens e para fazer as consultas dos endpoints. Espera um valor do tipo ***boolean***.
- ***hide***: Define se o atributo será ou não exibido no JSON de retorno. Espera um valor do tipo ***boolean***.
- ***unique***: Define se o atributo deve ser único na base de dados ou não. Caso o seu valor seja verdadeiro (*true*), o analisador de dados retornará um erro para a requisição, caso este atributo já exista. Espera um valor do tipo ***boolean***.
- ***nullable***: Por padrão, todos os atributos declarados nos metadados serão obrigatórios na inserção de novos registros. Caso este atributo estiver com o valor verdadeiro (*true*), o mesmo torna-se opcional. Espera um valor do tipo ***boolean***.
- ***max_length***: Declara a quantidade máxima de tamanho/caracteres que o campo suporta. Será utilizada posteriormente para barrar entradas maiores que esse valor no banco de dados. Espera um valor do tipo ***integer***.
- ***recommendable***: Define se o atributo será utilizado como base para as recomendações ou não. Será utilizado pelo motor de recomendações para levar em consideração apenas os atributos que o usuário da API julgar relevantes. Espera um valor do tipo ***boolean***.
- ***weight***: Se o atributo ***recommendable*** for verdadeiro (*true*), define o a peso do atributo nos cálculos de recomendação. O peso varia de um a dez. Espera um valor do tipo ***integer***.

Assim que a API toma conhecimento dos metadados, já é possível ao usuário da API preencher o banco de dados com seus usuários, itens e avaliações iniciais.

Por outro lado, a API, antes de persistir os metadados na base, adiciona alguns atributos para facilitar o gerenciamento posterior, dentre eles: um atributo de **data** para representar a data de inserção, um atributo de **versão** para identificar a evolução da estrutura e, por fim, um atributo ***active*** que, se possuir valor verdadeiro, determina qual é grupo de metadados que está atualmente em vigor. Uma linha do tempo dos metadados com as modificações feitas pelo usuário da API pode ser acessada através do endpoint **/metadata/:metadata_type/history**.

3.2 Modelos de Metadados, Itens, Usuários e Avaliações

Para gerenciar o fluxo de dados dentro da API, se fez necessária a criação de modelos encapsuladores, que tem como função abstrair a complexidade da validação, gerenciamento e representação destes dados dinâmicos.

Levando em conta que não se sabe quais serão os atributos de um item, por exemplo, antes do tempo de execução, é necessário que a validação seja feita com base não nos atributos do item, mas sim nos atributos previamente definidos pelos metadados do mesmo. Dessa forma, é possível contrastar o padrão esperado, definido nos metadados, com os dados efetivamente enviados pelo usuário da API.

Levando em consideração os atributos do grupo de metadados apresentados na seção Padrão de Metadados, a figura 12 apresenta um exemplo de validação, feito na linguagem Python, utilizando tais atributos para dinamicamente verificar o padrão do item recebido através de uma requisição do usuário da API. As partes irrelevantes do código foram omitidas para melhor apresentação.

Figura 12 – Validação baseada nos metadados.

```
for attr in self.meta.attributes:
    if attr['name'] not in self.item:
        if 'nullable' not in attr or not attr['nullable']:
            raise StatusCodeException(
                'Missing %s attribute' % attr['name'],
                400
            )
    elif type(self.item[attr['name']]).__name__ != attr['type']:
        raise StatusCodeException(
            '%s attribute has wrong type' % attr['name'],
            400
        )
return True
```

Fonte: O Autor.

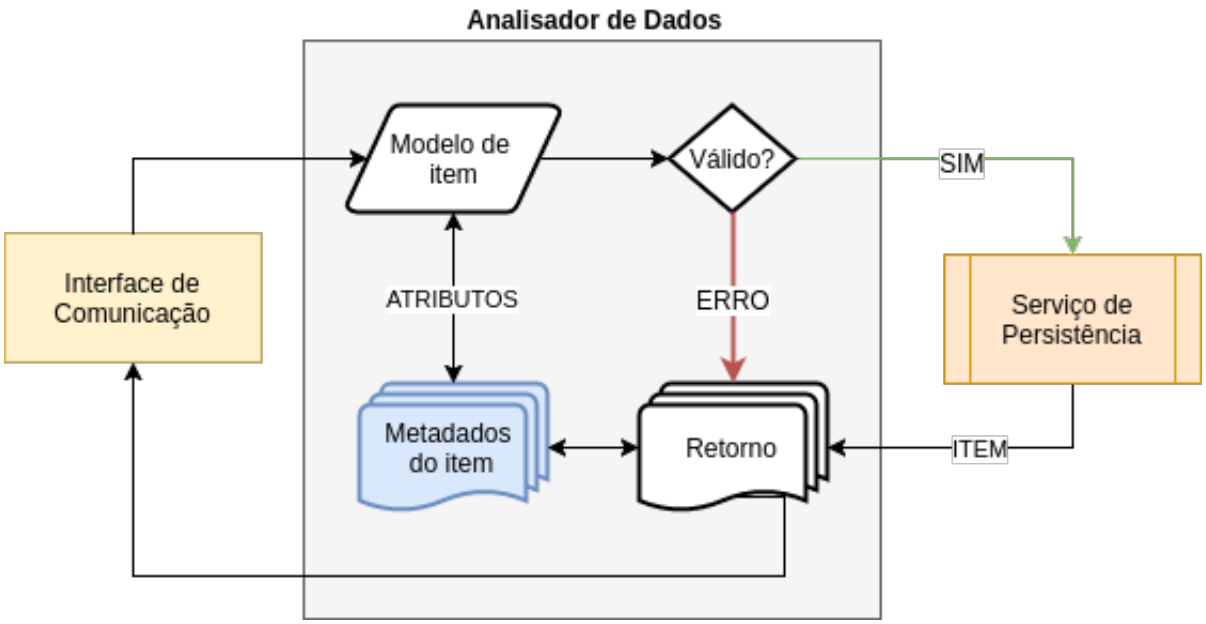
Ao contrário das aplicações comuns, que delegam o trabalho de manter a saúde dos dados para a base de dados, esta estratégia não é viável nesta aplicação devido ao fato da estrutura do banco de dados não só ser dinâmica, como também totalmente customizada pelo usuário da API.

Dessa forma, o trecho de código mostrado na figura 12, em conjunto outras classes auxiliares, tem como objetivo garantir que, apenas os dados enviados dentro do padrão descrito no grupo de metadados correspondente, serão persistidos no banco de dados.

Sendo assim, a lógica da figura pode ser descrita da seguinte forma: primeiro a aplicação percorre todos os atributos dos metadados correspondente. Para cada atributo descrito nos metadados, a aplicação testa se este atributo, caso seja obrigatório (***nullable:false***), está contido no JSON recebido pelo *endpoint*. Se o atributo obrigatório não existir, a aplicação retorna um JSON de erro com o *status code* igual a **400**, indicando uma ***bad request***. Caso o atributo exista, a aplicação testa se o tipo do dado recebido condiz com o tipo declarado nos metadados (***type***). Uma vez que todos os atributos obrigatórios são satisfeitos pelo JSON recebido, a aplicação o persiste na base de dados.

Para ilustrar o processo, a figura 13 demonstra um exemplo do fluxo interno do **analisador de dados**, desde o recebimento da informação, até a persistência e posterior retorno da mesma. O código completo dos modelos de dados pode ser visto no apêndice Código Fonte da API.

Figura 13 – Esquema do fluxo do analisador de dados.



Fonte: O Autor.

Representando o fluxo de modificação dos dados de um item específico, nota-se na figura 13, que o analisador de dados recebe as devidas requisições através da interface de

comunicação, transformando-as em modelos dos recursos. Assim que os modelos são construídos, são carregados da base de dados os metadados vigentes, utilizados posteriormente no contraste e validação destes modelos. Caso o modelo de item, representação abstrata da requisição do usuário da API, seja válido, o mesmo é persistido na base de dados e, posteriormente, enviado como retorno. Caso o item não seja válido, ao invés de persisti-lo na base, uma mensagem de erro é enviada como retorno ao usuário da API.

Todavia, em caso de sucesso na persistência, antes de efetivamente retornar o *feedback* ao agente da requisição, o analisador de dados constrói a estrutura de retorno a partir dos atributos dos metadados em vigor. O atributo declarado como identificador único (***key:true***) nos metadados é obrigatoriamente adicionado na estrutura de retorno, uma vez que é utilizado como parâmetro base em todos os *endpoints* relacionados. Além disso, o atributo ***hide*** define se o atributo existente será exibido ou não no JSON de retorno.

3.3 Persistência

A base de um sistema de recomendação é o conhecimento prévio, uma vez que, independente da métrica ou método utilizados, todos utilizam do cruzamento de informações conhecidas para gerar predições (vide seção 2). Dessa forma, é necessário que alguma estratégia eficiente de armazenamento e consulta seja utilizada, tendo em vista que o cruzamento dessas informações pode gerar matrizes excedendo milhões linhas e colunas (GOMEZ-URIBE; HUNT, 2016).

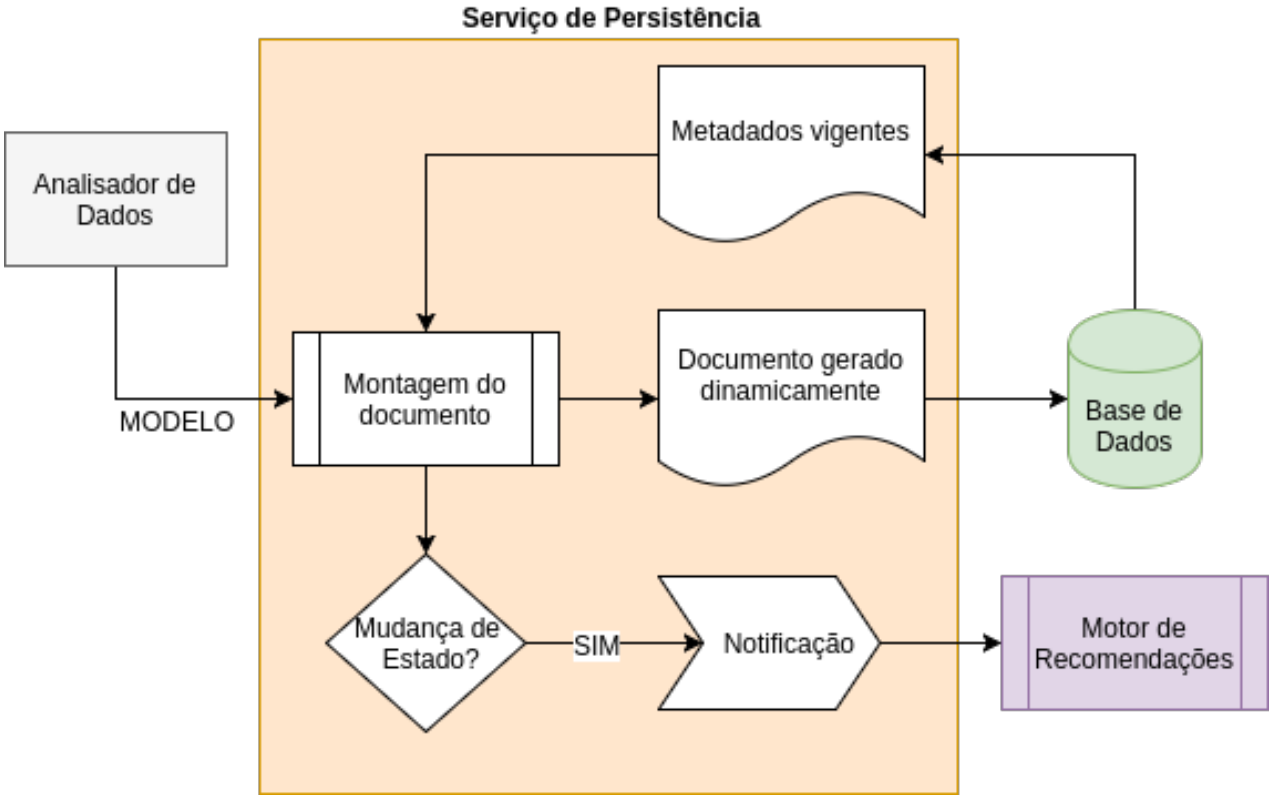
Visando utilizar uma estratégia de persistência que se destacasse no processamento de grandes conjuntos de dados, ponto essencial para boas recomendações, a API implementa uma persistência utilizando um banco de dados não relacional (NoSQL) orientado a documentos (LEAVITT, 2010), que além de proficiente no manejo dos dados, também persiste os mesmos em estruturas baseadas em JSON, como apresentado por Padhy, Patra e Satapathy (2011).

Devido ao JSON ser o padrão escolhido para a estrutura de dados da aplicação, a similaridade entre o documento armazenado no banco de dados e o JSON de retorno ao usuário da API é muito pequena, reduzindo o nível de complexidade como um todo. Além disso, é permitida a inserção de documentos dinâmicos, sem um padrão previamente definidos em tabelas e campos, sendo possível inserir novos atributos em itens, usuários e avaliações apenas declarando-os nos metadados, sem a necessidade de modificar a estrutura da base de dados.

A fim de abstrair este processo para os outras partes da aplicação, foram criadas serviços de persitência para cada módulo existente, responsáveis por administrar toda a comunicação com o banco de dados. Tais serviços são utilizados tanto pelo **analisador**

de dados quando pelo **motor de recomendações**, seja para a persistência de novas recomendações recém geradas ou para novos itens enviados pelo usuário da API através da interface de comunicação. A figura 14 demonstra o processo de construção dinâmica do item através do modelo e sua persistência no banco de dados.

Figura 14 – Esquema do serviço de persistência.



Fonte: O Autor.

Como apresentado na figura 14, o serviço de persistência recebe um modelo, previamente criado e validado pelo **analisador de dados**. Este modelo será contrastado com os metadados vigentes para geração dinâmica de um documento no formato necessário para sua inserção no banco de dados. Assim como nos *endpoints* da interface de comunicação, os serviços de persitência usam como parâmetros para administração de documentos específicos o identificador único (***key:true***) declarado nos metadados, gerando documentos com base neste identificador.

Além da persistência do modeo recebido, o serviço de persistência também é encarregado de notificar o motor de recomendações de quaisquer modificações ou adições aos documentos existentes. Esta notificação é necessária para que o motor de recomendações saiba quando é necessário treinar novos documentos recém adicionados, ou quando treinar novamente documentos que mudaram seu estado. O processo de recebimento das notificações e treinamento da base pode ser visto na seção O Motor de Recomendações.

4 O Motor de Recomendações

Toda a estrutura acima descrita tem como objetivo possibilitar o funcionamento do motor de recomendações, fornecendo os dados necessários para a geração das matrizes de predição. O motor de recomendações, por sua vez, tem como objetivo efetivamente gerar as recomendações para quaisquer usuários ou itens, levando em consideração o padrão de gostos do usuário, o conteúdo do item e a relação entre ambos.

Do mesmo modo que o analisador de dados, o motor de recomendações faz uso dos modelos de usuários, itens e avaliações visando obter informações geradas dinamicamente com o uso dos metadados. Dentre as informações relevantes para o motor de recomendações estão:

- **Atributos recomendáveis**: Os atributos marcados com a *tag* ***recommendable:true*** mapeam quais atributos devem ser levados em consideração pelos algoritmos.
- **Peso dos atributos**: Os atributos marcados com a *tag* ***weight*** mapeam quais atributos, além de recomendáveis, recebem um peso diferenciado para os algoritmos, maximizando o nível de personalização.
- **Identificadores únicos**: Os atributos marcados com a *tag* ***key:true*** identificam quais informações serão utilizadas como identificadores de itens e usuários durante a execução dos algoritmos, permitindo sua correta persistência, além de manter a integridade das recomendações. Além disso, permitem identificar quais usuários avaliaram quais itens, informação essencial para a recomendação colaborativa.
- **Valor da avaliação**: Os atributos dos metadados de avaliação marcados com a *tag* ***rating*** fornecem o valor de cada avaliação do usuário aos itens, permitindo a identificação do padrão de gostos do usuário, essencial para o cálculo da similaridade entre os itens.

De posse destes dados persistidos na base, o motor de recomendações consegue cruzá-los a fim de obter duas matrizes: a matriz de nível de similaridade entre itens e, posteriormente, a matriz de itens de maior similaridades não avaliados para cada usuário. Essas matrizes são posteriormente desmembradas em rakings de itens e persistidos em cada usuário e item, com seus itens de maior similaridade, em outras palavras, suas recomendações.

Como forma de recomendação será utilizado o método híbrido, composto dos métodos **baseado em conteúdo** e **filtragem colaborativa**, escalonando através do método com melhor precisão momentânea. Dessa forma é possível atender qualquer tipo

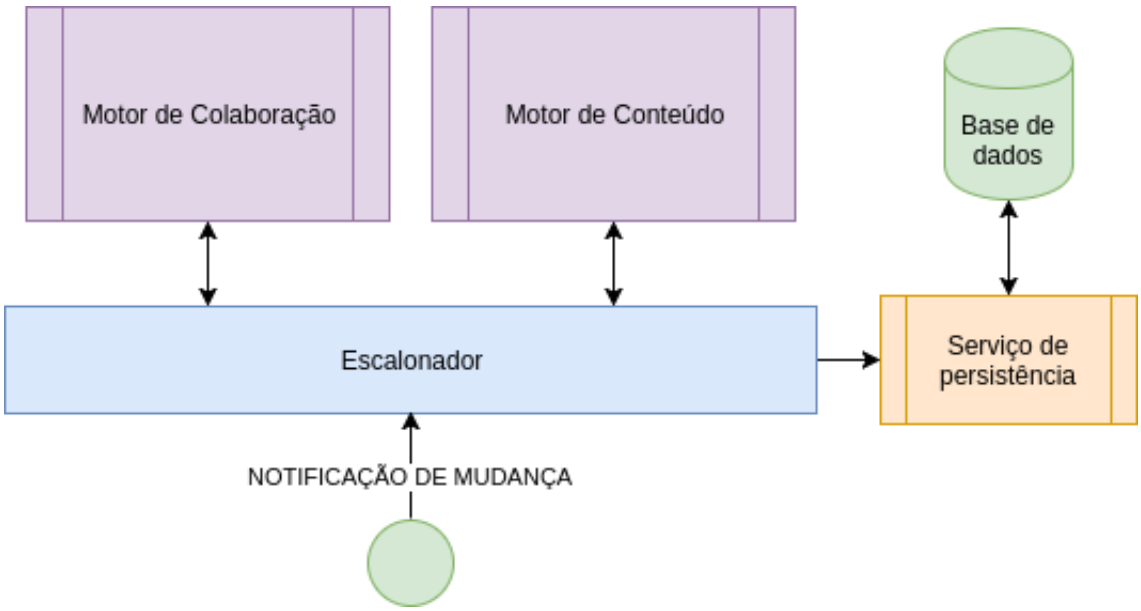
de metadado, fornecendo recomendações independente do número de usuários, itens e avaliações na base de dados.

Em um primeiro momento, a API utilizará o método baseado em conteúdo para fornecer as recomendações e predições, uma vez que poucos itens estarão avaliados e o método colaborativo não terá modelos de usuários suficientes. Ao passo que as métricas de relações entre usuários e quantidade de modelos processados sejam supridas, a API passará a utilizar o método de filtragem colaborativa, unindo os resultados com os maiores níveis de similaridades, levando em conta as duas diferentes abordagens.

Visando salientar de forma concisa o processo de recomendações, centro deste trabalho, esta seção está divida em três partes. Primeiramente é abordada a interação entre o motor de recomendações e as demais funcionalidades da API, o modo como são notificadas as mudanças e o assincronismo entre as requisições do usuário e a geração das recomendações. Em seguida, são detalhados os dois módulos que compõem o motor de recomendações: o **motor de conteúdo** e o **motor colaborativo**, abordando suas diferenças e peculiaridades.

4.1 Escalonador

Figura 15 – Funcionamento do escalonador de recomendações.



Fonte: O Autor.

Levando em consideração o nível de processamento necessário para a geração de recomendações precisas, a aplicação as excuta de forma paralela a requisição das recomendações, visando ocultar a complexidade do processo para o usuário da API.

sem comprometer o tempo de resposta deste tipo de requisição ao usuário da API, foi necessário paralelizar o processo de geração das recomendações,

4.2 Motor de Conteúdo

Conforme abordado na seção Método Baseado em Conteúdo, os sistemas de recomendação baseados em conteúdo analisam o conteúdo de itens previamente avaliados pelo usuário. Esta seção aborda a implementação deste tipo de sistema recomendador no contexto da aplicação, dissecando os principais tópicos relacionados a sistemas baseados em conteúdo e como foram implementados de modo a serem personalizados através dos metadados.

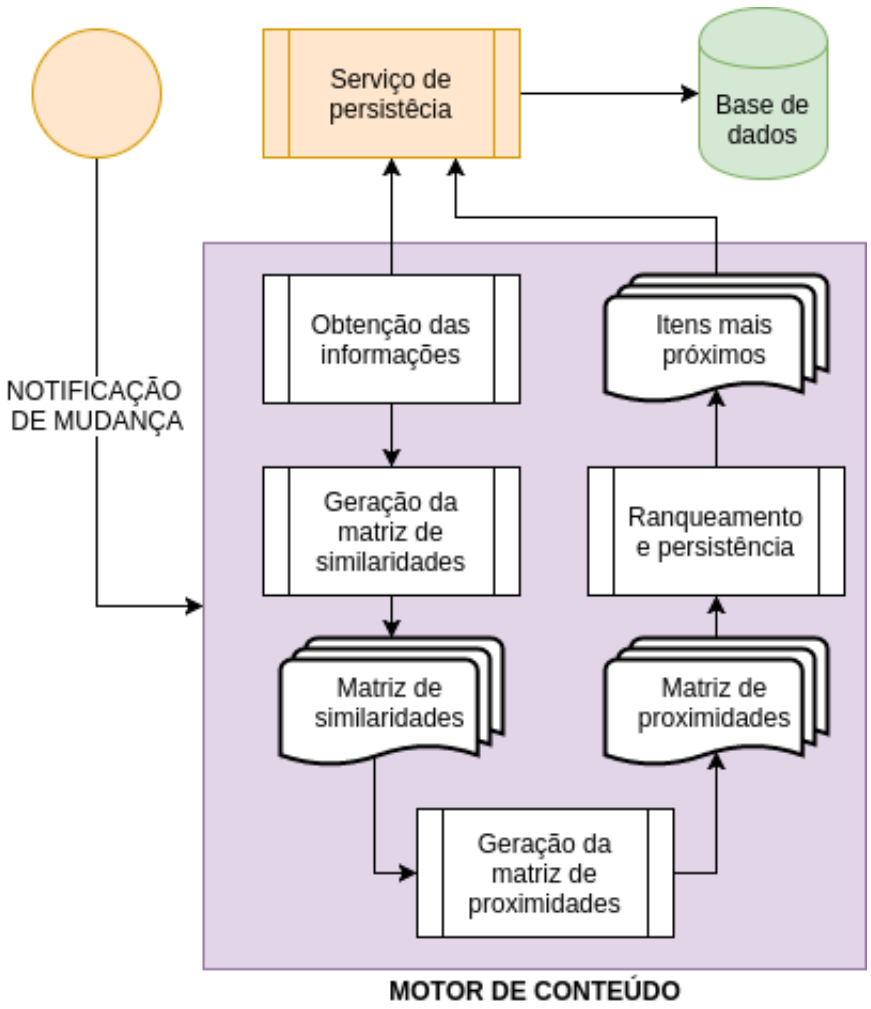
Para que seja possível fornecer recomendações de itens baseando-se em seu conteúdo de forma dinâmica, assim que o motor de recomendações é notificado de uma mudança relevante no estado dos dados, como apresentado na seção Escalonador, uma série de passos são executados. Este processo pós-notificação pode ser divido em cinco partes:

1. **Obtenção das informações**: carrega uma lista de todos os itens com os atributos a serem processados (***recommendable***) declarados nos metadados.
2. **Cálculo da similaridade**: para cada item I_m obtido, gera uma matriz de $[m, n]$ itens com o grau de similaridade entre o item corrente e os demais.
3. **Cálculo da proximidade**: para cada item I_n da matriz de similaridades, calcula a distância de coseno entre o item I_n e cada item da lista $\{S_0, ..., S_j\}$, sendo S um item similar a I_n.
4. **Persistência por proximidade**: ordena a lista de distâncias, resultante de cada item I_n, da menor distância para a maior, persistindo um ranking dos cinquenta itens mais próximos como similares do item I_n na base de dados.
5. **Obtenção das recomendações**: consulta a base de dados para obter os k itens da lista de similares para o item I_n e os retorna ao usuário da API.

Como resultado final do processamento feito pelo **motor de conteúdo**, as recomendações não são retornadas ao usuário da API, mas sim persistidas no banco de dados. Isso ocorre pois o tempo de retorno se tornaria muito alto, devido a quantidade de processos a serem feitos, além do principal fato de que as notificações de mudança são recebidas pelo motor em momentos diferentes das requisições de recomendação do usuário da API. O processo de geração das recomendações, abstraído da interação com o escalonador, pode ser visto na figura 16.

Como

Figura 16 – Funcionamento do motor de conteúdo.



Fonte: O Autor.

Exemplificar os 5 passos com uma figura e em momentos temporais diferentes.

Explicar primeiro o processo, como o motor funciona Explicar depois como o recomendador recomenda.

4.3 Motor Colaborativo

Explicar primeiro o processo, como o motor funciona Explicar depois como o recomendador recomenda.

5 Testes e Implementação

5.1 *Seed* de Dados

5.2 Documentação

5.3 Instalação e Inicialização

5.4 Tecnologias Utilizadas

4 RESULTADOS E DISCUSSÃO

1 Trabalhos Futuros

5 CONSIDERAÇÕES FINAIS

Devido ao grande número de implementações dos sistemas de recomendação nas mais diversas áreas que aqui foram apresentadas, torna-se notável a vasta gama de aplicações das técnicas e, mais do que isso, a necessidade de um serviço multipropósito desprendido do uso de linguagens de programação específicas. Sendo assim, ao final do cronograma, espera-se a obtenção de uma API que satisfaça estes requisitos.

Além disso, é importante ressaltar preocupação na construção de uma documentação direta e coesa ao longo de todo o processo, visando uma futura colaboração externa da comunidade, sendo a API de código aberto. Desta forma, o trabalho pode servir não só como uma alternativa *open-source* aos sistemas de recomendação, mas também como uma tecnologia de utilização simples para futuros estudos na área.

Referências

ADOMAVICIUS, Gediminas; TUZHILIN, Alexander. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. **IEEE transactions on knowledge and data engineering**, IEEE, v. 17, n. 6, p. 734–749, 2005. Citado 8 vezes nas páginas 9, 10, 14, 15, 16, 18, 19 e 23.

_____. Context-aware recommender systems. In: **Recommender systems handbook**. [S.l.]: Springer, 2015. p. 191–226. Citado na página 15.

BALABANOVIĆ, Marko; SHOHAM, Yoav. Fab: content-based, collaborative recommendation. **Communications of the ACM**, ACM, v. 40, n. 3, p. 66–72, 1997. Citado 2 vezes nas páginas 22 e 23.

BENNETT, James; LANNING, Stan *et al.* The netflix prize. In: NEW YORK, NY, USA. **Proceedings of KDD cup and workshop**. [S.l.], 2007. v. 2007, p. 35. Citado na página 10.

BREESE, John S; HECKERMAN, David; KADIE, Carl. Empirical analysis of predictive algorithms for collaborative filtering. In: MORGAN KAUFMANN PUBLISHERS INC. **Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence**. [S.l.], 1998. p. 43–52. Citado na página 18.

BROZOVSKY, Lukas; PETRICEK, Vaclav. Recommender system for online dating service. **arXiv preprint cs/0703042**, 2007. Citado 2 vezes nas páginas 10 e 12.

BURKE, Robin. Hybrid recommender systems: Survey and experiments. **User modeling and user-adapted interaction**, Springer, v. 12, n. 4, p. 331–370, 2002. Citado 3 vezes nas páginas 19, 20 e 21.

_____. Hybrid web recommender systems. **The adaptive web**, Springer, p. 377–408, 2007. Citado na página 19.

CARBONELL, Jaime G; MICHALSKI, Ryszard S; MITCHELL, Tom M. An overview of machine learning. In: **Machine learning**. [S.l.]: Springer, 1983. p. 3–23. Citado na página 13.

CLAYPOOL, Mark *et al.* Combining content-based and collaborative filters in an online newspaper. In: CITESEER. **Proceedings of ACM SIGIR workshop on recommender systems**. [S.l.], 1999. v. 60. Citado 2 vezes nas páginas 22 e 23.

COPPIN, Ben. **Inteligência artificial**. [S.l.]: Grupo Gen-LTC, 2015. Citado 2 vezes nas páginas 9 e 13.

CROCKFORD, Douglas. The application/json media type for javascript object notation (json). 2006. Citado na página 31.

FIELDING, Roy *et al.* **Hypertext transfer protocol–HTTP/1.1**. [S.l.], 1999. Citado na página 27.

GAVALAS, Damianos *et al.* Mobile recommender systems in tourism. **Journal of Network and Computer Applications**, Elsevier, v. 39, p. 319–333, 2014. Citado na página 10.

GOMEZ-URIBE, Carlos A; HUNT, Neil. The netflix recommender system: Algorithms, business value, and innovation. **ACM Transactions on Management Information Systems (TMIS)**, ACM, v. 6, n. 4, p. 13, 2016. Citado 2 vezes nas páginas 14 e 39.

GUO, Guibing *et al.* Librec: A java library for recommender systems. In: **UMAP Workshops**. [S.l.: s.n.], 2015. Citado na página 12.

HILL, Will *et al.* Recommending and evaluating choices in a virtual community of use. In: ACM PRESS/ADDISON-WESLEY PUBLISHING CO. **Proceedings of the SIGCHI conference on Human factors in computing systems**. [S.l.], 1995. p. 194–201. Citado na página 9.

HUANG, Zan *et al.* A graph-based recommender system for digital library. In: ACM. **Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries**. [S.l.], 2002. p. 65–73. Citado na página 10.

LEAVITT, Neal. Will nosql databases live up to their promise? **Computer**, IEEE, v. 43, n. 2, 2010. Citado na página 39.

LINDEN, Greg; SMITH, Brent; YORK, Jeremy. Amazon. com recommendations: Item-to-item collaborative filtering. **IEEE Internet computing**, IEEE, v. 7, n. 1, p. 76–80, 2003. Citado na página 10.

LOPS, Pasquale; GEMMIS, Marco De; SEMERARO, Giovanni. Content-based recommender systems: State of the art and trends. In: **Recommender systems handbook**. [S.l.]: Springer, 2011. p. 73–105. Citado 2 vezes nas páginas 16 e 17.

MCNEE, Sean M; RIEDL, John; KONSTAN, Joseph A. Being accurate is not enough: how accuracy metrics have hurt recommender systems. In: ACM. **CHI'06 extended abstracts on Human factors in computing systems**. [S.l.], 2006. p. 1097–1101. Citado na página 24.

MLADENIC, Dunja. Text-learning and related intelligent agents: a survey. **IEEE intelligent systems and their applications**, IEEE, v. 14, n. 4, p. 44–54, 1999. Citado na página 16.

NASCIMENTO, Vinicius Dalto do. **FILTRAGEM COLABORATIVA COMO SERVIÇO UTILIZANDO PROCESSAMENTO NA GPU**. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2013. Citado na página 12.

PADHY, Rabi Prasad; PATRA, Manas Ranjan; SATAPATHY, Suresh Chandra. Rdbms to nosql: Reviewing some next-generation non-relational databases. **International Journal of Advanced Engineering Science and Technologies**, v. 11, n. 1, p. 15–30, 2011. Citado na página 39.

PANG, Bo; LEE, Lillian; VAITHYANATHAN, Shivakumar. Thumbs up?: sentiment classification using machine learning techniques. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. **Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10**. [S.l.], 2002. p. 79–86. Citado na página 14.

PAZZANI, Michael; BILLSUS, Daniel. Content-based recommendation systems. **The adaptive web**, Springer, p. 325–341, 2007. Citado na página 17.

RESNICK, Paul; VARIAN, Hal R. Recommender systems. **Communications of the ACM**, ACM, v. 40, n. 3, p. 56–58, 1997. Citado na página 14.

RICCI, Francesco; ROKACH, Lior; SHAPIRA, Bracha. **Introduction to recommender systems handbook**. [S.l.]: Springer, 2011. Citado 3 vezes nas páginas 10, 14 e 15.

RODRIGUEZ, Alex. Restful web services: The basics. **IBM developerWorks**, 2008. Citado na página 26.

RUSSELL, Stuart; NORVIG, Peter. **Inteligência artificial**. [S.l.]: Elsevier, 2004. Citado 2 vezes nas páginas 9 e 13.

SARWAR, Badrul *et al.* Item-based collaborative filtering recommendation algorithms. In: ACM. **Proceedings of the 10th international conference on World Wide Web**. [S.l.], 2001. p. 285–295. Citado na página 18.

SCHAFER, JHJB *et al.* Collaborative filtering recommender systems. **The adaptive web**, Springer, p. 291–324, 2007. Citado 2 vezes nas páginas 17 e 19.

SCHAFER, J Ben; KONSTAN, Joseph; RIEDL, John. Recommender systems in e-commerce. In: ACM. **Proceedings of the 1st ACM conference on Electronic commerce**. [S.l.], 1999. p. 158–166. Citado 2 vezes nas páginas 14 e 15.

SCHAFER, J Ben; KONSTAN, Joseph A; RIEDL, John. E-commerce recommendation applications. In: **Applications of Data Mining to Electronic Commerce**. [S.l.]: Springer, 2001. p. 115–153. Citado na página 10.

SEBASTIANI, Fabrizio. Machine learning in automated text categorization. **ACM computing surveys (CSUR)**, ACM, v. 34, n. 1, p. 1–47, 2002. Citado na página 14.

TRAN, Thomas; COHEN, Robin. Hybrid recommender systems for electronic commerce. In: **Proc. Knowledge-Based Electronic Markets, Papers from the AAAI Workshop, Technical Report WS-00-04, AAAI Press**. [S.l.: s.n.], 2000. Citado 2 vezes nas páginas 21 e 22.

APÊNDICE A – Código Fonte da API

O módulo **app.py** é responsável por declarar todos os *endpoints* (recursos) da interface de comunicação que estarão acessíveis pelo usuário da API.

```
# Global defines
app = FlaskAPI(__name__)
app.config.from_pyfile('../settings.py')
app.url_map.converters['objectid'] = ObjectIDConverter
api = Api(app)

api.add_resource(Token, Token.ENDPOINT, endpoint='token')
api.add_resource(User, User.ENDPOINT, endpoint='user')
api.add_resource(Users, Users.ENDPOINT, endpoint='users')
api.add_resource(Recommend, Recommend.ENDPOINT, endpoint='recommend')
api.add_resource(RecommendPagination, RecommendPagination.ENDPOINT,
    endpoint='recommend_pagination')
api.add_resource(Train, Train.ENDPOINT, endpoint='train')
api.add_resource(TrainItem, TrainItem.ENDPOINT, endpoint='train_item')
api.add_resource(Items, Items.ENDPOINT, endpoint='items')
api.add_resource(Item, Item.ENDPOINT, endpoint='item')
api.add_resource(Metadata, Metadata.ENDPOINT, endpoint='metadata')
api.add_resource(MetadataList, MetadataList.ENDPOINT, endpoint='
    metadata_list')
api.add_resource(Rating, Rating.ENDPOINT, endpoint='rating')
api.add_resource(Ratings, Ratings.ENDPOINT, endpoint='ratings')

@app.route('/', methods=['GET'])
def root():
    return {'status': 'running'}
```

1 Metadados

```
class UserMetadata(object):
    """
    This class creates and validates a given user metadata dict to store it
    as internal attributes.

    Args:
        - metadata (dict): An user metadata dict representation.
        - version (int): The current version of user metadata
    representation.
        - active (boolean): A boolean to tell if the given dict is the
    current active in database.
```

```
10      """
11
12      def __init__(self, metadata, version=1, active=False):
13          if not metadata:
14              raise StatusCodeException('User metadata not found', 404)
15
16          self.meta = metadata
17          self.type = self.meta['type']
18          self.attributes = self.meta['attributes']
19          self.active = self.meta['active'] if 'active' in self.meta else
    active
20          self.created_at = self.meta['created_at'] if 'created_at' in self.
    meta else datetime.now()
21          self.version = self.meta['version'] if 'version' in self.meta else
    version
22
23          if self.type != 'user':
24              raise StatusCodeException('Invalid type', 400)
25
26          if self.attributes:
27              for attribute in self.attributes:
28                  if 'name' not in attribute:
29                      raise StatusCodeException('Missing name attribute at
    user metadata', 400)
30                  elif 'type' not in attribute:
31                      raise StatusCodeException('Missing type for "%s" at
    user metadata' % attribute['name'], 400)
32          else:
33              raise StatusCodeException('Missing attributes for user metadata
    ',
34                                        400)
35
36      def get_required_attributes(self):
37          """
38          Return all user metadata required attributes.
39
40          It iterates over the metadata attributes finding all required ones.
41
42          Returns:
43              An array with all user required attributes
44
45          """
46          return [
47              attribute for attribute in self.attributes
48              if 'nullable' not in attribute or not attribute['nullable']
49          ]
50
```

```
51      def get_recommendable_attributes(self):
52          """
53          Return all user metadata recommendable attributes.
54
55          It iterates over the metadata attributes finding all recommendable
    ones.
56
57          Returns:
58              An array with all user recommendable attributes
59
60          """
61          return [
62              attribute['name'] for attribute in self.attributes
63              if 'recommendable' in attribute and attribute['recommendable']
    and attribute['type'] == 'string'
64          ]
65
66      def to_database(self):
67          """
68          Builds a BSON representation of the metadata to be stored in
    mongoDB database.
69
70          It uses Metadata class attributes to buil a BSON object.
71
72          Returns:
73              A BSON dict to be stored.
74          """
75          return {
76              'type': self.type,
77              'attributes': self.attributes,
78              'active': self.active,
79              'created_at': self.created_at,
80              'version': self.version
81          }
82
83      def to_json(self):
84          """
85          Builds a JSON representation of the metadata to be returned as API
    output to user.
86
87          It uses Metadata class attributes to build a JSON object.
88
89          Returns:
90              A JSON dict to be returned as API output.
91          """
92          return {
93              'type': self.type,
```

```
            'attributes': self.attributes,
            'active': self.active,
            'created_at': self.created_at,
            'version': self.version
        }
```

```
class ItemMetadata(object):
    """
    This class creates and validates a given item metadata dict to store it
    as internal attributes.

    Args:
        - metadata (dict): An item metadata dict representation.
        - version (int): The current version of item metadata
    representation.
        - active (boolean): A boolean to tell if the given dict is the
    current active in database.
    """

    def __init__(self, metadata, version=1, active=False):
        if not metadata:
            raise StatusCodeException('Item metadata not found', 404)

        self.meta = metadata
        self.type = self.meta['type']
        self.attributes = self.meta['attributes']
        self.active = self.meta['active'] if 'active' in self.meta else
    active
        self.created_at = self.meta[
            'created_at'] if 'created_at' in self.meta else datetime.now()
        self.version = self.meta[
            'version'] if 'version' in self.meta else version

        if self.type != 'item':
            raise StatusCodeException('Invalid type', 400)

        if self.attributes:
            for attribute in self.attributes:
                if 'name' not in attribute:
                    raise StatusCodeException(
                        'Missing name attribute at item metadata', 400)
                elif 'type' not in attribute:
                    raise StatusCodeException(
                        'Missing type for "%s" at item metadata' %
                        attribute['name'], 400)
        else:
            raise StatusCodeException('Missing attributes for item metadata
    ',
```

```
                                    400)

    def get_required_attributes(self):
        """
        Return all item metadata required attributes.

        It iterates over the metadata attributes finding all required ones.

        Returns:
            An array with all item required attributes
        """
        return [
            attribute for attribute in self.attributes if 'nullable' not in attribute or not attribute['nullable']
        ]

    def get_recommendable_attributes(self):
        """
        Return all item metadata recommendable attributes.

        It iterates over the metadata attributes finding all recommendable ones.

        Returns:
            An array with all item recommendable attributes
        """
        return [
            attribute['name'] for attribute in self.attributes
            if 'recommendable' in attribute and attribute['recommendable'] and attribute['type'] == 'string'
        ]

    def to_database(self):
        """
        Builds a BSON representation of the metadata to be stored in mongoDB database.

        It uses Metadata class attributes to buil a BSON object.

        Returns:
            A BSON dict to be stored.
        """
        return {
            'type': self.type,
            'attributes': self.attributes,
            'active': self.active,
            'created_at': self.created_at,
```

```
            'version': self.version
        }

    def to_json(self):
        """
        Builds a JSON representation of the metadata to be returned as API output to user.

        It uses Metadata class attributes to build a JSON object.

        Returns:
            A JSON dict to be returned as API output.
        """
        return {
            'type': self.type,
            'attributes': self.attributes,
            'active': self.active,
            'created_at': self.created_at,
            'version': self.version
        }
```

```
class RatingMetadata(object):
    """
    This class creates and validates a given rating metadata dict to store it
    as internal attributes.

    Args:
        - metadata (dict): An rating metadata dict representation.
        - version (int): The current version of rating metadata representation.
        - active (boolean): A boolean to tell if the given dict is the current active in database.
    """

    def __init__(self, metadata, version=1, active=False):
        if not metadata:
            raise StatusCodeException('Rating metadata not found', 404)

        self.meta = metadata
        self.type = self.meta['type']
        self.attributes = self.meta['attributes']
        self.active = self.meta['active'] if 'active' in self.meta else active
        self.created_at = self.meta['created_at'] if 'created_at' in self.meta else datetime.now()
        self.version = self.meta['version'] if 'version' in self.meta else version
```

```
22
23          if self.type != 'rating':
24              raise StatusCodeException('Invalid type', 400)
25
26          if self.attributes:
27              for attribute in self.attributes:
28                  if 'name' not in attribute:
29                      raise StatusCodeException('Missing name attribute at
    item metadata', 400)
30                  elif 'type' not in attribute:
31                      raise StatusCodeException('Missing type for "%s" at
    item metadata' %
32                                attribute['name'], 400)
33          else:
34              raise StatusCodeException('Missing attributes for item metadata
    ', 400)
35
36      def get_required_attributes(self):
37          """
38          Return all item metadata required attributes.
39
40          It iterates over the metadata attributes finding all required ones.
41
42          Returns:
43              An array with all rating required attributes
44
45          """
46          return [
47              attribute for attribute in self.attributes if 'nullable' not in
     attribute or not attribute['nullable']
48          ]
49
50      def get_recommendable_attributes(self):
51          """
52          Return all rating metadata recommendable attributes.
53
54          It iterates over the metadata attributes finding all recommendable
    ones.
55
56          Returns:
57              An array with all rating recommendable attributes
58
59          """
60          return [
61              attribute['name'] for attribute in self.attributes
62              if 'recommendable' in attribute and attribute['recommendable']
    and attribute['type'] == 'string'
```

```
63          ]
64
65      def to_database(self):
66          """
67          Builds a BSON representation of the metadata to be stored in
    mongoDB database.
68
69          It uses Metadata class attributes to buil a BSON object.
70
71          Returns:
72              A BSON dict to be stored.
73          """
74          return {
75              'type': self.type,
76              'attributes': self.attributes,
77              'active': self.active,
78              'created_at': self.created_at,
79              'version': self.version
80          }
81
82      def to_json(self):
83          """
84          Builds a JSON representation of the metadata to be returned as API
    output to user.
85
86          It uses Metadata class attributes to build a JSON object.
87
88          Returns:
89              A JSON dict to be returned as API output.
90          """
91          return {
92              'type': self.type,
93              'attributes': self.attributes,
94              'active': self.active,
95              'created_at': self.created_at,
96              'version': self.version
97          }
```

2 Modelos de Estruturas

```
1 class UserModel(object):
2     def __init__(self, user):
3         self.meta_service = UserMetadataService()
4         self.meta = UserMetadata(self.meta_service.get_active())
5         self.user = user
6
7     def _get_existent_attributes(self):
```

```
8         return [attr for attr in self.meta.attributes
9             if (attr['name'] in self.user) and  ('hide' not in attr)]
10
11    def validate(self):
12        for attr in self.meta.attributes:
13            if attr['name'] not in self.user:
14                if 'nullable' not in attr or not attr['nullable']:
15                        raise StatusCodeException('Missing %s attribute' %
    attr['name'], 400)
16            elif get_synonymous(type(self.user[attr['name']]).__name__) !=
    attr['type']:
17                raise StatusCodeException('%s attribute has wrong type' %
    attr['name'], 400)
18        return True
19
20    def set_id(self, user_id):
21        self.user['_id'] = user_id
22
23    def to_database(self):
24        item = {}
25        for attr in self.meta.attributes:
26            if attr['name'] in self.user:
27                item[attr['name']] = self.user[attr['name']]
28        return item
29
30    def to_json(self):
31        json = {
32            '_id': ObjectIDConverter.to_url(self.user['_id'])
33        }
34        for attr in self._get_existent_attributes():
35            json[attr['name']] = self.user[attr['name']]
36        return json
```

```
1 class ItemModel(object):
2     """
3     This class creates, manage and validates a given item dict to store it
4     in database and ouput it to all requests.
5
6     Args:
7         - item (dict): An item dict representation.
8     """
9
10    def __init__(self, item):
11        self.meta_service = ItemMetadataService()
12        self.meta = ItemMetadata(self.meta_service.get_active())
13        self.item = item
14
15    def _get_existent_attributes(self):
```

```
16          return [
17              attr for attr in self.meta.attributes
18              if (attr['name'] in self.item) and ('hide' not in attr)
19          ]
20
21      def validate(self):
22          """
23          Validates existent item attributes based on current
24          item metadata.
25
26          It uses item metadata definitions to validate item
27          dynamic attributes and values.
28
29          Returns:
30              ``True`` if all item attributes match with metadata. Otherwise,
31              throw an Exception to eatch type of validation.
32          """
33          for attr in self.meta.attributes:
34              if attr['name'] not in self.item:
35                  if 'nullable' not in attr or not attr['nullable']:
36                          raise StatusCodeException('Missing %s attribute' %
    attr['name'], 400)
37              elif get_synonymous(type(self.item[attr['name']]).__name__) !=
    attr['type']:
38                  raise StatusCodeException('%s attribute has wrong type' %
    attr['name'], 400)
39          return True
40
41      def to_database(self):
42          """
43          Builds a BSON representation of the item to be stored in database.
44
45          It uses Item class existent attributes to build a BSON object.
46
47          Returns:
48              A BSON dict to be stored.
49          """
50          item = {}
51          for attr in self.meta.attributes:
52              if attr['name'] in self.item:
53                  item[attr['name']] = self.item[attr['name']]
54          return item
55
56      def set_id(self, item_id):
57          """
58          Sets user_id to Item class after it have been generated in database
    .
```

```
59
60          Args:
61              - item_id (objectId): The correspondent item_id in database.
62
63          Returns:
64              None
65          """
66          self.item['_id'] = item_id
67
68      def to_json(self):
69          """
70          Builds a JSON representation of the item to be returned as API
    output to user.
71
72          It uses Item class existent attributes to build a JSON object.
73
74          Returns:
75              A JSON dict to be returned as API output.
76          """
77          json = {'_id': ObjectIDConverter.to_url(self.item['_id'])}
78          for attr in self._get_existent_attributes():
79              json[attr['name']] = self.item[attr['name']]
80          return json
81
82      def to_rec_json(self, similarity=0):
83          """
84          Builds a JSON representation of the item to be returned as API
85          recommendation utput.
86
87          It uses Item class existent attributes to build a JSON object
88          that contains item attributes and similarity level.
89
90          Returns:
91              A JSON dict to be returned as API output.
92          """
93          json = self.to_json()
94          json['similarity'] = similarity
95          return json
```

```
1 class RatingModel(object):
2     def __init__(self, rating):
3         self.rating_service = RatingMetadataService()
4         self.meta = RatingMetadata(self.rating_service.get_active())
5         self.rating = rating
6
7     def _get_existent_attributes(self):
8         return [
9             attr for attr in self.meta.attributes
```

```
            if (attr['name'] in self.rating) and ('hide' not in attr)
        ]

    def validate(self):
        for attr in self.meta.attributes:
            if attr['name'] not in self.rating:
                if 'nullable' not in attr or not attr['nullable']:
                        raise StatusCodeException('Missing %s attribute' %
attr['name'], 400)
            elif get_synonymous(type(self.rating[attr['name']]).__name__)
!= attr['type']:
                raise StatusCodeException('%s attribute has wrong type' %
attr['name'], 400)
        return True

    def to_database(self):
        rating = {}
        for attr in self.meta.attributes:
            if attr['name'] in self.rating:
                rating[attr['name']] = self.rating[attr['name']]
        return rating

    def set_id(self, rating_id):
        self.rating['_id'] = rating_id

    def to_json(self):
        json = {'_id': ObjectIDConverter.to_url(self.rating['_id'])}
        for attr in self._get_existent_attributes():
            json[attr['name']] = self.rating[attr['name']]
        return json

    def to_rec_json(self, similarity=0):
        json = self.to_json()
        json['similarity'] = similarity
        return json
```

3 Recursos de *endpoints*

```
class User(Resource):

    ENDPOINT = '/user/<objectid:user_id>'

    def __init__(self):
        self.service = UserService()

    @auth.middleware_auth_token
    def get(self, user_id):
```

```
10          try:
11              user = self.service.get_by_id(user_id)
12              user = UserModel(user)
13              return make_response(user.to_json())
14          except StatusCodeException as ex:
15              return ex.to_response()
16          except Exception as ex:
17              return StatusCodeException(ex.message, 500).to_response()
18
19      @auth.middleware_auth_token
20      def put(self, user_id):
21          try:
22              user = UserModel(request.get_json())
23              if user.validate():
24                  user = UserModel(self.service.update(user_id, user.
    to_database()))
25                  return make_response(user.to_json())
26              else:
27                  raise StatusCodeException('User not found', 404)
28          except StatusCodeException as ex:
29              return ex.to_response()
30          except Exception as ex:
31              return StatusCodeException(ex.message, 500).to_response()
32
33      @auth.middleware_auth_token
34      def delete(self, user_id):
35          try:
36              if self.service.get_by_id(user_id):
37                  self.service.remove(user_id)
38                  return make_response()
39              else:
40                  raise StatusCodeException('User not found', 404)
41          except StatusCodeException as ex:
42              return ex.to_response()
43          except Exception as ex:
44              return StatusCodeException(ex.message, 500).to_response()
45
46
47  class Users(Resource):
48
49      ENDPOINT = '/user'
50
51      def __init__(self):
52          self.service = UserService()
53
54      @auth.middleware_auth_token
55      def get(self):
```

```
56          all_users = self.service.get_all()
57          json_users = [UserModel(user).to_json() for user in all_users]
58          return make_response(json_users)
59
60      def post(self):
61          try:
62              user = UserModel(request.get_json())
63              if user.validate():
64                  user_id = self.service.insert(user.to_database())
65                  user.set_id(user_id) # XXX - Generate user recommendations
66                  return make_response(user.to_json())
67              else:
68                  raise StatusCodeException('Conflict', 409)
69          except StatusCodeException as ex:
70              return ex.to_response()
71          except Exception as ex:
72              return StatusCodeException(ex.message, 500).to_response()
```

```
1  class Item(Resource):
2
3      ENDPOINT = '/item/<objectid:item_id>'
4
5      def __init__(self):
6          self.item_service = ItemService()
7
8      @auth.middleware_auth_token
9      def get(self, item_id):
10         try:
11             item = self.item_service.get_by_id(item_id)
12             if item:
13                 return make_response(ItemModel(item).to_json())
14             else:
15                 raise StatusCodeException('Item not found', 404)
16         except StatusCodeException as ex:
17             return ex.to_response()
18         except Exception as ex:
19             return StatusCodeException(ex.message, 500).to_response()
20
21     @auth.middleware_auth_token
22     def delete(self, item_id):
23         try:
24             if self.item_service.get_by_id(item_id):
25                 self.item_service.remove(item_id)
26                 return make_response()
27             else:
28                 raise StatusCodeException('Item not found', 404)
29         except StatusCodeException as ex:
30             return ex.to_response()
```

```
        except Exception as ex:
            return StatusCodeException(ex.message, 500).to_response()


class Items(Resource):

    ENDPOINT = '/item'

    def __init__(self):
        self.item_service = ItemService()

    @auth.middleware_auth_token
    def get(self):
        all_items = self.item_service.get_all()
        return make_response([ItemModel(item).to_json() for item in
    all_items])

    @auth.middleware_auth_token
    def post(self):
        try:
            item = ItemModel(request.get_json())
            if item.validate():
                item_id = self.item_service.insert(item.to_database())
                item.set_id(item_id)
                ContentWorker().train_item(item_id)
                return make_response(item.to_json())
            else:
                raise StatusCodeException('Conflict', 409)
        except StatusCodeException as ex:
            return ex.to_response()
        except Exception as ex:
            return StatusCodeException(ex.message, 500).to_response()
```

```
class Rating(Resource):

    ENDPOINT = '/rating/<objectid:rating_id>'

    def __init__(self):
        self.service = RatingService()

    @auth.middleware_auth_token
    def get(self, rating_id):
        try:
            rating = self.service.get_by_id(rating_id)
            rating = RatingModel(rating)
            return make_response(rating.to_json())
        except StatusCodeException as ex:
            return ex.to_response()
```

```
16          except Exception as ex:
17              return StatusCodeException(ex.message, 500).to_response()
18
19      @auth.middleware_auth_token
20      def put(self, rating_id):
21          try:
22              rating = self.service.update(rating_id, request.get_json())
23              if rating:
24                  rating = RatingModel(rating)
25                  return make_response(rating.to_json())
26              else:
27                  raise StatusCodeException('Rating not found', 404)
28          except StatusCodeException as ex:
29              return ex.to_response()
30          except Exception as ex:
31              return StatusCodeException(ex.message, 500).to_response()
32
33      @auth.middleware_auth_token
34      def delete(self, rating_id):
35          try:
36              if self.service.get_by_id(rating_id):
37                  self.service.remove(rating_id)
38                  return make_response()
39              else:
40                  raise StatusCodeException('Rating not found', 404)
41          except StatusCodeException as ex:
42              return ex.to_response()
43          except Exception as ex:
44              return StatusCodeException(ex.message, 500).to_response()
45
46
47  class Ratings(Resource):
48
49      ENDPOINT = '/rating'
50
51      def __init__(self):
52          self.service = RatingService()
53
54      @auth.middleware_auth_token
55      def get(self):
56          all_ratings = self.service.get_all()
57          json_ratins = [RatingModel(rating).to_json() for rating in
    all_ratings]
58          return make_response(json_ratins)
59
60      def post(self):
61          try:
```

```
            rating = request.get_json()
            if not False: # XXX - Metadata validation here!
                rating['_id'] = self.service.insert(rating)
                return make_response(rating)
            else:
                raise StatusCodeException('Conflict', 409)
        except StatusCodeException as ex:
            return ex.to_response()
        except Exception as ex:
            return StatusCodeException(ex.message, 500).to_response()
```

```
class Recommend(Resource):

    ENDPOINT = '/recommend/<objectid:item_id>/top/<int:
    number_of_recommendations>'

    def __init__(self):
        self.recommender_service = RecommenderService()

    @auth.middleware_auth_token
    def get(self, item_id, number_of_recommendations=10):
        try:
            return make_response(self.recommender_service.recommend(item_id
    , end=number_of_recommendations))
        except StatusCodeException as ex:
            return ex.to_response()
        except Exception as ex:
            return StatusCodeException(ex.message, 500).to_response()

class RecommendPagination(Resource):

    ENDPOINT = '/recommend/<objectid:item_id>/<int:start>/<int:end>'

    def __init__(self):
        self.recommender_service = RecommenderService()

    @auth.middleware_auth_token
    def get(self, item_id, start, end):
        try:
            return make_response(self.recommender_service.recommend(item_id
    , start, end))
        except StatusCodeException as ex:
            return ex.to_response()
        except Exception as ex:
            return StatusCodeException(ex.message, 500).to_response()
```

```
class TrainItem(Resource):

```

```
    ENDPOINT = '/train/<objectid:item_id>'

    @auth.middleware_auth_token
    def get(self, item_id):
        try:
            ContentWorker().train_item(item_id)
            return make_response() # XXX - Separar content worker do
    prepare pra validar
        except StatusCodeException as ex:
            return ex.to_response()
        except Exception as ex:
            return StatusCodeException(ex.message, 500).to_response()


class Train(Resource):

    ENDPOINT = '/train'

    @auth.middleware_auth_token
    def get(self):
        try:
            ContentWorker().train()
            CollaborativeWorker().train()
            return make_response()
        except StatusCodeException as ex:
            return ex.to_response()
        except Exception as ex:
            info(traceback.print_exc())
            return StatusCodeException(ex.message, 500).to_response()
```

```
class Token(Resource):

    ENDPOINT = '/auth/token'

    def post(self):
        try:
            email = request.data.get('email', None)
            password = request.data.get('password', None)
            if email and password:
                token = auth.generate_token(email, password)
                return make_response({'token': token})
            else:
                raise StatusCodeException('Invalid parameters', 400)
        except StatusCodeException as ex:
            return ex.to_response()
        except Exception as ex:
            return StatusCodeException(ex.message, 500).to_response()
```

```
class Metadata(Resource):

    ENDPOINT = '/metadata/<string:meta_type>'

    def __init__(self):
        self.item_meta_service = ItemMetadataService()
        self.user_meta_service = UserMetadataService()

    @auth.middleware_auth_token
    def post(self, meta_type):
        try:
            if meta_type == 'item':
                service = self.item_meta_service
                new_metadata = ItemMetadata(request.get_json(), version=1,
    active=True)
            elif meta_type == 'user':
                service = self.user_meta_service
                new_metadata = UserMetadata(request.get_json(), version=1,
    active=True)
            else:
                raise StatusCodeException('Invalid type', 400)

            if not service.get_active():
                service.insert(new_metadata.to_database())
                return make_response(new_metadata.to_json())
            else:
                raise StatusCodeException('Conflict', 409)
        except StatusCodeException as ex:
            return ex.to_response()
        except Exception as ex:
            return StatusCodeException(ex.message, 500).to_response()

    @auth.middleware_auth_token
    def put(self, meta_type):

        def _get_new_version(service):
            meta = service.get_active()
            if meta:
                return meta['version'] + 1

            raise StatusCodeException('Item metadata not found', 404)

        try:
            if meta_type == 'item':
                service = self.item_meta_service
                new_metadata = request.get_json()
                new_metadata = ItemMetadata(new_metadata, _get_new_version(
```

```
    service), True)
46              elif meta_type == 'user':
47                  service = self.user_meta_service
48                  new_metadata = request.get_json()
49                  new_metadata = UserMetadata(new_metadata, _get_new_version(
    service), True)
50              else:
51                  raise StatusCodeException('Invalid type', 400)
52
53              service.disable_all()
54              service.insert(new_metadata.to_database())
55              return make_response(new_metadata.to_json())
56
57          except StatusCodeException as ex:
58              return ex.to_response()
59          except Exception as ex:
60              return StatusCodeException(ex.message, 500).to_response()
61
62
63  class MetadataList(Resource):
64
65      ENDPOINT = '/metadata/<string:meta_type>/history'
66
67      def __init__(self):
68          self.item_meta_service = ItemMetadataService()
69          self.user_meta_service = UserMetadataService()
70
71      @auth.middleware_auth_token
72      def get(self, meta_type):
73          try:
74              if meta_type == 'item':
75                  json_metadata = [ItemMetadata(meta).to_json()
76                      for meta in self.item_meta_service.get_all()]
77              elif meta_type == 'user':
78                  json_metadata = [UserMetadata(meta).to_json()
79                      for meta in self.user_meta_service.get_all()]
80              else:
81                  raise StatusCodeException('Invalid type', 400)
82
83              return make_response(json_metadata)
84          except StatusCodeException as ex:
85              return ex.to_response()
86          except Exception as ex:
87              return StatusCodeException(ex.message, 500).to_response()
```

4 Serviços de Persistência

```
class ItemMetadataService(object):

    def __init__(self):
        self.item_meta = db.item_metadata

    def get_active(self):
        return self.item_meta.find_one({'active': True})

    def insert(self, item_meta_dict):
        return self.item_meta.insert(item_meta_dict)

    def disable_all(self):
        return self.item_meta.update({'active': True}, {'$set': {'active':
    False}})

    def get_all(self):
        return self.item_meta.find().sort([('version', pymongo.DESCENDING)
    ])
```

```
class ItemService(object):

    def __init__(self):
        self.items = db.items
        self.meta = ItemMetadata(ItemMetadataService().get_active())

    def get_by_id(self, item_id):
        return self.items.find_one({'_id': item_id})

    def get_all(self):
        return self.items.find({}, {'similar': 0})

    def get_info(self, item_list):
        return self.items.find({'_id': {'$in' : item_list}}, {'similar':
    0})

    def insert(self, item_dict):
        return self.items.insert(item_dict)

    def remove(self, item_id):
        return self.items.remove({'_id': item_id})

    def update_recommendations(self, item_id, recommendations):
        return self.items.find_one_and_update(
            {'_id': item_id},
            {'$set': {'similar': recommendations}}
        )

    def get_rec_data(self):
```

```
        attributes = self.meta.get_recommendable_attributes()
        coalesce_attributes = {'$project': {}}
        concat_filter = {'$project': {}}

        for attr in attributes:
            coalesce_attributes['$project'][attr] = {
                '$ifNull': ['${name}'.format(name=attr), '']}

        concat_filter['$project']['concated_attrs'] = {
            '$concat': ['${name}'.format(name=attr) for attr in attributes
    ]}

        return self.items.aggregate([coalesce_attributes, concat_filter])
```

```
class RatingMetadataService(object):

    def __init__(self):
        self.rating_meta = db.rating_metadata

    def get_active(self):
        return self.rating_meta.find_one({'active': True})

    def insert(self, rating_meta_dict):
        return self.rating_meta.insert(rating_meta_dict)

    def disable_all(self):
        return self.rating_meta.update({'active': True}, {'$set': {'active'
    : False}})

    def get_all(self):
        return self.rating_meta.find().sort([('version', pymongo.DESCENDING
    )])
```

```
class RatingService(object):

    def __init__(self):
        self.ratings = db.ratings

    def get_by_id(self, rating_id):
        return self.ratings.find_one({'_id': rating_id})

    def get_all(self):
        return self.ratings.find()

    def update(self, rating_id, new_data):
        return self.ratings.find_one_and_update(
            {'_id': rating_id},
            {'$set': {
```

```
                'name': new_data['name'],
                'age': new_data['age'],
                'email': new_data['email']
            }},
            return_document=ReturnDocument.AFTER
        )

    def insert(self, rating_dict):
        return ObjectIDConverter.to_url(self.ratings.insert(rating_dict))

    def remove(self, rating_id):
        return self.ratings.remove({'_id': rating_id})
```

```
class UserMetadataService(object):

    def __init__(self):
        self.user_meta = db.user_metadata

    def get_active(self):
        return self.user_meta.find_one({'active': True})

    def insert(self, user_meta_dict):
        return self.user_meta.insert(user_meta_dict)

    def disable_all(self):
        return self.user_meta.update({'active': True}, {'$set': {'active':
    False}})

    def get_all(self):
        return self.user_meta.find().sort([('version', pymongo.DESCENDING)
    ])
```

```
class UserService(object):

    def __init__(self):
        self.users = db.users

    def get_by_token(self, token):
        return self.users.find_one({'auth.token': token})

    def get_by_id(self, user_id):
        return self.users.find_one({'_id': user_id})

    def get_all(self):
        return self.users.find()

    def expire_token(self, user_id, token):
        return self.users.find_one_and_update(
```

```
17              {'_id': user_id},
18              {'$set': {
19                  'auth': {'token': token, 'valid': False}
20              }}
21          )
22
23      def update(self, user_id, new_data):
24          return self.users.find_one_and_update(
25              {'_id': user_id},
26              {'$set': {
27                  'name': new_data['name'],
28                  'age': new_data['age'],
29                  'email': new_data['email']
30              }},
31              return_document=ReturnDocument.AFTER
32          )
33
34      def insert(self, user_dict):
35          return self.users.insert(user_dict)
36
37      def remove(self, user_id):
38          return self.users.remove({'_id': user_id})
39
40      def remove_token(self, user_id):
41          return self.users.find_one_and_update(
42              {'_id': user_id},
43              {'$set': {
44                  'auth': {}
45              }}
46          )
47
48      def exists(self, email):
49          return self.users.find_one({'email': email})
50
51      def verify(self, email, password):
52          user = self.users.find_one({'email': email, 'password': password})
53          if user:
54              return user['_id']
55          return False
56
57      def update_token(self, user_id, new_token):
58          return self.users.find_one_and_update(
59              {'_id': user_id},
60              {'$set': {
61                  'auth': {'token': new_token, 'valid': True}
62              }}
63          )
```

```
class RecommenderService(object):
    def __init__(self):
        self.item_service = ItemService()
        self.ratings = RatingService()

    def recommend(self, item_id, start=0, end=10):
        """
        Couldn't be simpler! Just retrieves the similar items and their 'score' from redis.

        :param item_id: int
        :param number_of_recommendations: number of similar items to return
        :return: A list of lists like: [["19", 0.2203], ["494", 0.1693], ...]. The first item in each sub-list is
        the item ID and the second is the similarity score. Sorted by similarity score, descending.
        """
        item = self.item_service.get_by_id(item_id)
        if item:
            if 'similar' in item:
                similar_items = item['similar'][start:end]
                similar_ids = [it['_id'] for it in item['similar'][start:end]]
                recommendations = self.item_service.get_info(similar_ids)
                json_recs = []
                for rec in recommendations:
                    for similar_item in similar_items:
                        if similar_item['_id'] == rec['_id']:
                            json_recs.append(ItemModel(rec).to_rec_json(similar_item['similarity']))
                json_recs = sorted(json_recs, cmp=lambda x,y: cmp(x['similarity'], y['similarity']), reverse=True) #Resort by similarity level
                return json_recs
            return {}
        else:
            raise StatusCodeException('Item not found', 404)

    def get_shared_preferences(self, user_A, user_B):
        """
        Returns the intersection of ratings for two users
        """
        ratings = self.ratings.get_all()
        if user_A not in ratings:
            raise KeyError("Couldn't find user '%s' in data" % user_A)
        if user_B not in ratings:
            raise KeyError("Couldn't find user '%s' in data" % user_B)
```

```
41
42          moviesA = set(ratings[user_A].keys())
43          moviesB = set(ratings[user_B].keys())
44          shared = moviesA & moviesB # Intersection operator
45
46          # Create a reviews dictionary to return
47          shared_prefs = {}
48          for item_id in shared:
49              shared_prefs[item_id] = (
50                  ratings[user_A][item_id]['rating'],
51                  ratings[user_B][item_id]['rating'],
52              )
53          return shared_prefs
```

5 Motores de Recomendação

```
1  from math import sqrt
2
3  def euclidean_distance(self, preferences):
4      """
5      Reports the Euclidean distance of two critics, A&B by
6      performing a J-dimensional Euclidean calculation of
7      each of their preference vectors for the intersection
8      of movies the critics have rated.
9      """
10     # Get the intersection of the rated titles in the data.
11
12     # If they have no rankings in common, return 0.
13     if len(preferences) == 0:
14         return 0
15
16     # Sum the squares of the differences
17     sum_of_squares = sum([pow(a-b, 2) for a, b in preferences.values()])
18
19     # Return the inverse of the distance to give a higher score to
20     # folks who are more similar (e.g. less distance) add 1 to prevent
21     # division by zero errors and normalize ranks in [0,1]
22     return 1 / (1 + sqrt(sum_of_squares))
23
24 def pearson_correlation(self, preferences):
25     """
26     Returns the Pearson Correlation of two user_s, A and B by
27     performing the PPMC calculation on the scatter plot of (a, b)
28     ratings on the shared set of critiqued titles.
29     """
30
31     # Store the length to save traversals of the len computation.
```

```
    # If they have no rankings in common, return 0.
    length = len(preferences)
    if length == 0:
        return 0

    # Loop through the preferences of each user_ once and compute the
    # various summations that are required for our final calculation.
    sumA = sumB = sumSquareA = sumSquareB = sumProducts = 0
    for a, b in preferences.values():
        sumA += a
        sumB += b
        sumSquareA += pow(a, 2)
        sumSquareB += pow(b, 2)
        sumProducts += a*b

    # Calculate Pearson Score
    numerator   = (sumProducts*length) - (sumA*sumB)
    denominator = sqrt(((sumSquareA*length) - pow(sumA, 2)) * ((sumSquareB*
    length) - pow(sumB, 2)))

    # Prevent division by zero.
    if denominator == 0:
        return 0

    return abs(numerator / denominator)
```

```
class ContentEngine(object):
    """
    This class creates the tfidf engine that will train item
    recommendations
    based on it's content.

    It creates a pandas dataframe containing all recommendable string data.
    Based on the created dataframe, a tfidf matrix will de made. After all,
    the cosine similarity array will be generated using the distance
    between axis
    in the tfidf matrix.
    """

    def __init__(self):
        start = time.time()
        self.item_service = ItemService()
        self.data = pd.DataFrame(list(self.item_service.get_rec_data()))
        self.tfidf = TfidfVectorizer(
            analyzer='word',
            ngram_range=(1, 3),
            min_df=0,
            smooth_idf=False,
```

```
21              stop_words='english')
22          self.tfidf_matrix = self.tfidf.fit_transform(
23              self.data['concated_attrs'])
24          self.cosine_similarities = linear_kernel(
25              self.tfidf_matrix, self.tfidf_matrix)
26          info("Training data ingested in %s seconds." % (time.time() - start
    ))
27
28      def _get_item_index(self, item_id):
29          for index, item in self.data.iterrows():
30              if item['_id'] == item_id:
31                  return item, index
32
33      def _train_item(self, item, index):
34          similar_indices = self.cosine_similarities[index].argsort()
    [:-50:-1]
35          recs = [(self.cosine_similarities[index][similar_item], self.data['
    _id'][similar_item])
36                  for similar_item in similar_indices]
37
38          recs = [
39              {
40                  '_id': item_id,
41                  'similarity': similarity
42              } for similarity, item_id in recs[1:]  # First item is the item
     itself, so remove it.
43          ]
44
45          self.item_service.update_recommendations(item['_id'], recs)
46
47      def train(self):
48          """
49          Train the engine.
50
51          Create a TF-IDF matrix of unigrams, bigrams, and trigrams for each
    product.
52
53          Then similarity is computed between all products using SciKit
    Cosine Similarity.
54
55          Iterate through each item's similar items and store the 50 most-
    similar.
56          Similarities and their scores are stored in database as a sorted
    set, with one set for each item.
57
58          Returns:
59              None
```

```
        """
        start = time.time()
        for index, item in self.data.iterrows():
            self._train_item(item, index)
        info("Engine trained in %s seconds." % (time.time() - start))

    def train_item(self, item_id):
        """
        Train the engine for a given item.

        Create a TF-IDF matrix of unigrams, bigrams, and trigrams for the given item.

        Then similarity is computed between the given product and all other products, using SciKit Cosine Similarity.

        Iterate through each item's similar items and store the 50 most-similar.
        Similarities and their scores are stored in database as a sorted set of the item.

        Args:
            - item_id (objectId): The item id from training.

        Returns:
            None

        """
        start = time.time()
        item, index = self._get_item_index(item_id)
        self._train_item(item, index)
        info("Item %s trained in %s seconds." %
             (item_id, (time.time() - start)))
```

```
class CollaborativeEngine(object):

    def __init__(self):
        start = time.time()
        self.item_service = ItemService()
        self.user_service = UserService()
        self.rating_service = RatingService()
        self.recommender_service = Recommender_service()

        self.items = pd.DataFrame(list(self.item_service.get_all()))
        self.users = pd.DataFrame(list(self.user_service.get_all()))
        self.ratings = pd.DataFrame(list(self.rating_service.get_all()))
        self.ds = Dataset.load_builtin(self.ratings)
        info("Training data ingested in %s seconds." % (time.time() - start
```

```
    ))
15
16      def train(self):
17          start = time.time()
18          self.calculatePearsonSimilarity(self.ratings, 1, 2)
19          #evaluate(algo, ds, measures=['RMSE', 'MAE'])
20          info("Engine trained in %s seconds." % (time.time() - start))
21
22      def similar_items(self, item, metric='euclidean', n=50):
23          # Metric jump table
24          metrics = {
25              'euclidean': metrics.euclidean_distance,
26              'pearson':   metrics.pearson_correlation,
27          }
28
29          distance = metrics.get(metric, None)
30          ratings = pd.DataFrame(list(self.rating_service.get_all()))
31
32          # Handle problems that might occur
33          if item not in ratings['movie_id']:
34              raise KeyError("Unknown item, '%s'." % item)
35          if not distance or not callable(distance):
36              raise KeyError("Unknown or unprogrammed distance metric '%s'."
    % metric)
37
38          similar_items = {}
39          for similar_item in ratings['movie_id']:
40              if similar_item == item:
41                  continue
42
43              similar_items[similar_item] = distance(self.recommender_service
    .get_shared_preferences(similar_item['user_id'], item['user_id']))
44
45          return heapq.nlargest(n, items.items(), key=itemgetter(1))
```