Um aplicativo mobile e sistema de recomendação para alinhamento entre skills e demandas de estágio em TI

Hugo Machado

RELATÓRIO DE PROJETO FINAL

CENTRO TÉCNICO CIENTÍFICO - CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de graduação em Ciência da Computação

Rio de Janeiro
Junho de 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Hugo Machado

Um aplicativo mobile e sistema de recomendação para alinhamento entre skills e demandas de estágio em TI

Relatório de Projeto Final, apresentado ao programa de Ciência da Computação da PUC-Rio como requisito parcial para a obtenção do Bacharel em Ciência de Computação.

Orientador: Prof. Markus Endler

Rio de Janeiro
Junho de 2021

Agradecimentos

Aos meus orientadores Markus Endler e Valeria de Paiva, pelo apoio, ensinamentos e orientação que possibilitaram a realização deste projeto.

Aos meu pais, por toda a dedicação, ajuda e companheirismo durante esta grande etapa.

Resumo

Endler, Markus; Endler, Markus. **Um aplicativo mobile e sistema de recomendação para alinhamento entre skills e demandas de estágio em TI**. Rio de Janeiro, 2021. 62p. Projeto de Graduação – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Alunos universitários de períodos iniciais ou até de períodos mais avançados - frequentemente encontram dificuldades para o ingresso no mercado de trabalho, por ainda estarem aprimorando os seus conhecimentos técnicos e suas competências profissionais. O objetivo deste projeto é desenvolver uma ferramenta de software que ajude alunos a terem seu primeiro contato com demandas reais. A ferramenta consiste de um serviço mobile que ajuda o estudante a divulgar as suas competências técnicas (skills) e a encontrar uma demanda/tarefa real de trabalho profissional em que possa trabalhar e ganhar experiência profissional, possivelmente de forma não remunerada. O aplicativo também permite que pequenos e médios empreendedores, assim como pesquisadores da própria universidade, possam descrever e ofertar demandas/tarefas, e obter indicações dos alunos "mais aptos" a trabalharem nelas. De modo a sugerir demandas que estejam mais próximas às competências de um estudante - usando o universo de skills que um perfil possa ter - esse projeto utiliza Natural Language Processing (NLP), que nos últimos anos tem sido usado com grande sucesso em sistemas de recomendação. Portanto, juntamente com o aplicativo mobile, foi criada também uma API de recomendação e de matching de skills-tasks integrada com o aplicativo, que denomino de *Aplicai*

Palavras-chave

Flutter; Mobile; Sistema de Recomendação. Word2Vec; Word Mover Distance.

Abstract

Endler, Markus; Endler, Markus (Advisor). . Rio de Janeiro, 2021. 62p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Students from initial semesters or even those who are in more advanced semesters - face certain barriers regarding their initial entry into the job market, because they are still improving their technical knowledge and professional skills. The goal of this project is to develop a software tool that helps students to have their first contact with the professional market. The tool consists of a mobile service that helps students to advertise their technical competencies (skills) and hence find a real professional demand/-task where they can work and gain professional experience, possibly unpaid. The application in question also aims to recommend and match skills and demands, so that enterprises can describe and offer tasks, and get suggestions of the students "best suited" to work on them. In order to suggest demands that are more adequate and closer to a student's competencies - using the universe of skills that a profile may have - this project uses Natural Language Processing (NLP), which in recent years has been used with great success in recommendation systems. So along with the mobile application, we also created a recommendation and skill-demand matching API integrated with the application, which we call *Aplicai*.

Keywords

Flutter; Mobile; Recomendation System; Word2Vec; Word Mover Distance.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Diante do atual mercado de trabalho existe uma enorme quantidade de informações no que tange à conhecimentos na área de computação. O seu processo evolutivo acontece de forma muito rápida, a cada dia novas tecnologias são criadas e evoluídas. De forma a estar sempre alinhado com essas tecnologias é muito importante praticar e se envolver com os temas atuais da área tecnológica. Alunos de graduação, muitas vezes, tendem a sentir mais dificuldade no momento de se inserirem no mercado de trabalho e sentem a necessidade de colocar em prática o que estão aprendendo na universidade. Aproveitando a motivação do aluno juntamente com o crescimento de demanda por trabalho especializado em função da expansão de novas empresas na área de tecnologia da informação, design e UX, o *Aplicai* poderá colaborar para o processo de formação dos alunos. De forma similar, o aplicativo propõe a encontrar sua primeira tarefa profissional com demandas concretas e reais, facilitando assim sua futura inserção no mercado de trabalho.

Atualmente, muitas redes sociais orientadas a negócios já estão utilizando-se de mecanismos conhecidos como push marketing (1) (diferentemente dos métodos mais tradicionais, como pull marketing, onde é o usuário que deve procurar a empresa / emprego) para impulsionar de forma automática anúncios de empregos. A proposta deste trabalho se baseia nessa estratégia, onde os usuários cadastrados no sistema poderão ter facilidade na hora de buscar uma demanda, já que sempre serão sugeridas, demandas que tenham mais correlações com as suas skills. Com isso, será necessário o desenvolvimento de uma plataforma, no caso, uma API, o qual será o sistema de recomendação. Em outras palavras, de forma macro, é um sistema de avaliação e priorização de afinidades (matchmaking).

Nos últimos anos sistemas de recomendação têm se tornado um atrativo muito grande para diversos usos. Um deles é na área de busca de emprego. Buscar ofertas de emprego atualmente com a evolução da internet se tornou muito simples, onde sites como Linkedin, Indeed, Vagas, etc... são exemplos de alguns mais populares, que com uma simples busca, é possível procurar ofertas de emprego de acordo com a necessidade. Alguns trabalhos como (2) (3) já demonstram a importância de usar skills como objeto principal de recomen-

dação. Baseado nesses trabalhos, a ideia do *Aplicai* é de fazer um bom uso de skills de usuários. Explorando sites como os mencionados acima, que ofertam empregos de acordo com a pesquisa, é possível observar que em grande parte, a busca é feita através do título da oferta e alguns outros atributos que ajudam na classificação. Existem diversas técnicas que podem ajudar a melhorar a busca de trabalhos usando sistemas de recomendação com machine learning.

O objetivo do *Aplicai* é um pouco diferente quando comparado a sites de oferta de emprego, uma vez que o foco é para alunos universitários e também que estes alunos se cadastrem em demandas de trabalho que tenham um prazo fixo e preestabelecido para acabar. Outra característica singular é que as demandas são para estágios não remunerados, já que a ideia é servir a alunos que têm como principal interesse ganhar experiência em desenvolvimento e assim aprimorar suas competências e adquirir também novas skills. O aplicativo realiza apenas a análise de adequação entre skills e ofertas, e possibilita o primeiro contato entre alunos e empreendedores. A comunicação posterior entre as partes é depois realizada da forma como for mais conveniente por ambas. De fato, um dos principais focos do trabalho é que a recomendação seja baseada em skills tratando o seu significado técnico e não apenas uma simples busca por palavras chaves que estejam presentes na descrição das demandas.

Portanto, o *Aplicai* visa o cadastramento dos usuários e de demandas com algumas informações cruciais, sendo uma das mais importantes, o cadastro de seus/suas skills. Alunos de uma universidade por exemplo, podem cadastrar uma skill que tenham aprendido no decorrer do curso ou por aprendizado próprio. Empreendedores já cadastrariam as demandas que seriam ofertadas, colocando as competências que são necessárias para que os alunos participem. Com esse conjunto de informações torna-se possível realizar um *matchmaking* entre ambas as partes usando um sistema que consiga capturar seus dados e usá-los com o objetivo de recomendar uma demanda de acordo com sua importância para um determinado aluno.

2
Objetivos do trabalho

Este projeto visa criar um sistema totalmente funcional. Serão utilizadas bibliotecas da linguagem Dart e Python além dos recursos do Firebase. Este trabalho pode ser dividido em duas partes do desenvolvimento:

1. Criação do aplicativo mobile/web
    - Desenvolvimento do Aplicativo mobile e também Web.
    - Integração com o Firebase
    - Desenvolvimento da UI de todas as telas
2. Criação do sistema de recomendação
    - Desenvolvimento do algoritmo de recomendação
    - Integração com o Firebase
    - Criação de uma API usando o framework Flask
    - Utilização a biblioteca Gensim para construir o algoritmo de recomendação

Para o funcionamento do sistema de recomendação este deve utilizar das skills que serão salvas pelo aplicativo de forma que possibilite a sugestão de demandas de acordo com um determinado usuário baseando-se em suas skills. Para que isso se torne possível, deve existir uma integração entre as duas partes mencionadas acima, de modo a formar um sistema mobile-cloud. A figura 2.1, representa a ideia do sistema de forma geral.

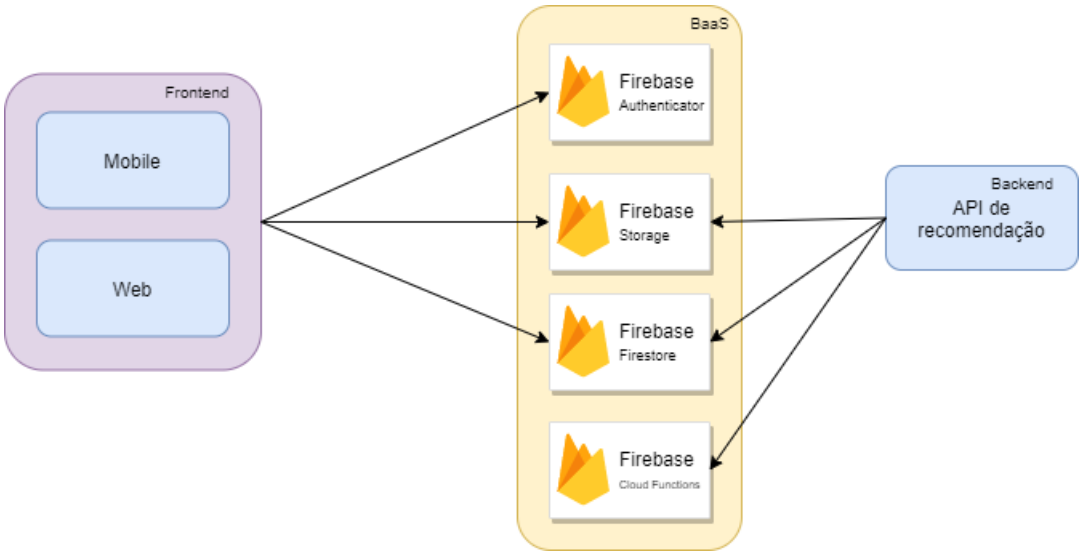

Figura 2.1: Arquitetura geral

3
Ferramentas e tecnologias utilizadas

Nesta seção são descritos os principais fundamentos, tecnologias e ferramentas utilizadas de forma a arquitetar a aplicação.

3.1
Firebase

Firebase [1] é hoje uma das maiores plataformas de Backend as a service (5). Atualmente temos o conceito de "as a service" como um conceito cada vez mais popular em desenvolvimento de software, principalmente para quem está iniciando a carreira como desenvolvedor ou trabalha em uma empresa menor, como uma startup. A plataforma disponibiliza atualmente, diversas funcionalidades que economizam muito o trabalho de um desenvolvedor. Através de recursos que ajudam a simplificar o desenvolvimento de um backend, como por exemplo seu uso para banco de dados. Esses fatores acabam por ter uma grande redução quanto ao custo de desenvolvimento - já que são necessários menos desenvolvedores - e grande parte do esforço ao desenvolver vários sistemas de backend pode ser deslocado para outras áreas. Por estas razões, a plataforma do Firebase foi adotada como um dos backends do *Aplicai*. Abaixo estão descritos todos os recursos do Firebase que foram adotados para a construção do sistema.

3.1.1
Firebase Authenticator

O Firebase Authenticator [2] é a ferramenta capaz de realizar cadastro e login de forma segura para os usuários da aplicação. O desenvolvimento de um backend de login e cadastro capaz de fornecer um sistema seguro para os usuários demanda bastante cuidado e tempo, pois dados sensíveis como uma senha por exemplo, podem ser comprometidos, levando então o sistema a ter diversas falhas de segurança. A utilização do Firebase Authenticator ajudou na integração de um sistema de cadastro, onde torna possível que cadastro e login dos clientes fiquem inteiramente sob a responsabilidade do Firebase,

[1] https://firebase.google.com/
[2] https://firebase.google.com/docs/auth

aliviando um grande esforço nessa parte. Um fator muito importante também é que o sistema de autenticação permite a integração com diferentes sistemas de cadastro, como por exemplo, login do Google, Facebook, e-mail, senha, etc... Dentro de todos os métodos disponíveis, o *Aplicai* faz uso do cadastro de e-mail e senha e também do login pelo Google. Mas, futuramente, pode facilmente incluir novos métodos através da fácil integração com o Firebase.

3.1.2 Firebase Firestore

O Firestore [3] é um banco de dados 'noSQL' que funciona como uma árvore de documentos JSON. Portanto, ele não trabalha com o conceito de tabelas como nos bancos 'SQL'. Uma das grandes vantagens do banco 'noSQL' é que estes possuem uma alta escalabilidade além de serem mais econômicos financeiramente. Isso acontece, pois grande parte das restrições, como o relacionamento entre tabelas, que existem normalmente em bancos relacionais, dificultam na escalabilidade horizontal, onde é mais comum escalar de forma vertical aumentando memória RAM e processador. O Firebase então, garante uma maior velocidade de desenvolvimento uma vez que não precisamos nos preocupar quanto a criar relações entre tabelas e também garante que um grande volume de dados pode ser lido ou gravado sem prejudicar a aplicação. Para a aplicação sua utilização principal é para armazenar dados de diversos usuários e demandas através da criação de diversos documentos com os dados importantes que precisam ser armazenados a longo prazo.

3.1.3 Firebase Storage

Firebase Firestore, é capaz de armazenar uma quantidade enorme de dados, porém dependendo do tamanho dos documentos pode levar a lentidão no banco de dados. Por exemplo, converter uma imagem em bytes e armazenar no Firestore não faz sentido. Por esse motivo na aplicação foi utilizada o Firebase Storage [4], com o intuito de salvar principalmente imagens e arquivos mais pesados que poderiam acarretar problemas durante buscas no banco de dados. Outro fator importante é que oferece uma segurança e gerenciamento para que apenas aplicações ou usuários autenticados tenham acesso ao conteúdo armazenado, fornecendo uma alta velocidade de download e upload.

[3] https://firebase.google.com/docs/firestore
[4] https://firebase.google.com/docs/storage

3.1.4 Firebase Cloud Functions

Cloud Functions [5] é uma plataforma serverless (4), onde funções são acionadas por eventos. Uma de suas vantagens é que não será necessário manter nenhum servidor e nem gerenciá-lo, tudo ficará do lado do Firebase, sendo necessário apenas o desenvolvimento do código. Existem várias formas para ativar essas funções. Para essa aplicação, a plataforma é utilizada para detectar alterações no banco de dados e então executar uma lógica de acordo com o tipo de evento.

3.2 Flutter

Flutter [6] é o framework escolhido para a construção da aplicação. Ele dispõe de um SDK que contém uma série de ferramentas que ajudam no desenvolvimento de aplicações. Um dos principais recursos do Flutter são os Widgets, que são componentes já pré-prontos que são altamente customizáveis e que possibilitam a criação rápida de uma interface para a aplicação. Um dos pontos mais interessantes é o fato do Flutter ser multiplataforma, onde com apenas um único código é possível compilar para Android, IOS, Desktop (diferentes sistemas operacionais) e também para Web. A versão para mobile é a que se encontra em um estado mais estável. A versão Web já é considerada pronta para produção, tendo algumas instabilidades quando comparada a versão mobile. A versão para Desktop ainda está em fase beta. Portanto, para o *Aplicai* foram geradas versões para Mobile e Web por serem mais estáveis.

3.3 BLOC Pattern

No desenvolvimento de uma aplicação frontend, separar a lógica da parte da interface do usuário, é fundamental para que seu desenvolvimento ocorra de forma organizada. O padrão BLOC foi apresentado na conferência Google I/O em 2018, visando trazer para o Flutter uma arquitetura que facilitasse gerir o estado e o acesso a dados de forma centralizada. A imagem abaixo mostra uma arquitetura simplificada usando BLOC. Os eventos são gerados pela camada da interface do usuário, onde cada evento é gerido por cada Bloc. Dentro desta camada existem mudanças de um estado para outro e é nessa etapa onde poderá ocorrer comunicação com agentes externos como uma API REST ou mesmo uma camada de repositório, onde existem transações com

[5] https://firebase.google.com/docs/functions
[6] flutter.dev

um banco de dados. Durante o desenvolvimento inicial do *Aplicai*, tivemos bastante problema de bugs, principalmente na parte da interface e também muita repetição de código. Com o uso da arquitetura BLOC, foi possível fazer um melhor tratamento de cada evento o que facilitou a interação da lógica com os diferentes Widgets do Flutter.

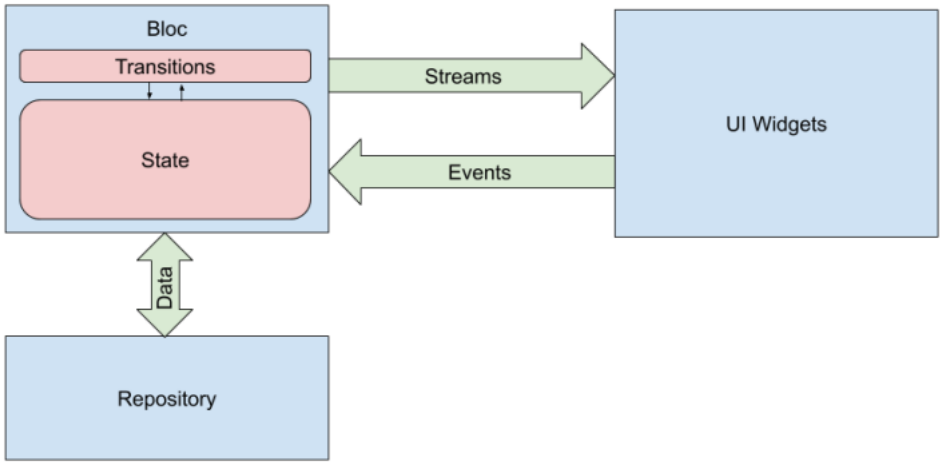

Figura 3.1: BLOC

3.4 Flask

Flask é um micro-framework web desenvolvido em Python. Por se tratar de um micro-framework, possui apenas funcionalidades consideradas essenciais. Esse framework é capaz de criar uma API REST. Ele foi escolhido para que pudesse trabalhar junto com o algoritmo de recomendação que foi desenvolvido na linguagem Python.

4
Fundamentos

Nesta seção serão resumidos dois dos principais fundamentos teóricos que serviram de base para a construção do algoritmo de recomendação.

4.1
Método Word2Vec

Word2Vec [1] é um dos métodos mais conhecidos na área de processamento de linguagem natural. Desenvolvido em 2013 por Tomas Mikolov, esse método consegue criar um word embedding de palavras de forma eficiente, através de um corpus de texto, usando redes neurais. Word embeddings são vetores com n dimensões capazes de representar o conceito de uma palavra. Dessa forma, usando cálculos matemáticos, existem diversas técnicas para calcular a distância entre esses vetores de palavras e assim, é possível verificar um grau de similaridade entre os pares. Essa técnica foi utilizada para a construção do sistema de recomendação. Através das skills obtidas tanto de demandas quanto de perfis de alunos, tornou-se possível a criação de um word embedding específico para o *Aplicai*, onde é possível analisar a real similaridade entre as skills de cada usuário com as demandas da aplicação.

O Word2Vec é composto por uma rede neural com duas camadas, que após o treinamento possibilita gerar um word embedding. Quando temos um grande corpus de texto, ao treinar utilizando essa técnica, o método consegue gerar vetores capazes de detectar palavras que estão ocorrendo dentro de um contexto similar. São dois os tipos de modelos que podem ser usados no Word2Vec, são eles o CBOW e o Skip-Gram.

- O modelo do CBOW consegue através do contexto prever qual é a palavra que está no meio, dentro de um conjunto de palavras.
- O modelo Skip-gram é o oposto do CBOW. Ele consegue prever qual é o contexto a partir de uma palavra.

Para entender melhor o funcionamento de uma rede neural para o modelo do Word2Vec, vamos tomar o exemplo da frase “Java é a melhor linguagem de

[1] https://en.wikipedia.org/wiki/Word2vec

programação". Se queremos prever por exemplo a palavra "linguagem", devemos então colocar como input da rede neural as palavras que estão entre "linguagem" que é no caso "melhor" e "programação", e o "de" é excluído por ser uma preposição (stopword). Como não é possível usar palavras de fato em uma rede neural, devemos realizar um processo de one-hot encoding, transformando cada palavra em um vetor com apenas 0 e 1. Vamos desconsiderar as stopwords como "a", "e" e "de". Assim temos a seguinte lista de vetores, que são agora o input da rede neural:

- Java=[1,0,0,0]
- melhor=[0,1,0,0]
- linguagem=[0,0,1,0]
- programação=[0,0,0,1]

No exemplo a seguir vamos simular um caso com CBOW. A rede deve ter apenas uma única hidden layer, onde ficam os vetores que representam os pesos que serão calculados. A multiplicação da matriz dos inputs junto com os pesos da hidden layer, resulta em um outro vetor de output. O tamanho da matriz de pesos é o que determina qual vai ser a dimensão do nosso word embedding gerado. Como temos quatro palavras, vamos tomar por exemplo um peso W gerado randomicamente e um vetor V de output gerado randomicamente, como costuma ser feito nos algoritmos de rede neural. I é o nosso input de quatro palavras. Então temos I x W com dimensão 4 x 2 e W x V com dimensão 2 x 4. Abaixo segue o exemplo descrito:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{X} \begin{bmatrix} 0.2 & 0.5 \\ 0.5 & 0.1 \\ 0.3 & 0.9 \\ 0.24 & 0.7 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.5 \\ 0.5 & 0.1 \\ 0.3 & 0.9 \\ 0.24 & 0.7 \end{bmatrix}$$ I x W é exatamente

o próprio W. Logo como resultado do output vamos ter W x V:

$$\begin{bmatrix} 0.2 & 0.5 \\ 0.5 & 0.1 \\ 0.3 & 0.9 \\ 0.24 & 0.7 \end{bmatrix} \text{X} \begin{bmatrix} 0.3 & 0.2 & 0.8 & 0.1 \\ 0.2 & 0.5 & 0.1 & 0.9 \end{bmatrix}$$

O output da rede neural é o resultado de W x V, gerando uma matriz 4

x 4. O output agora deve passar por uma activation function [2], transformando o resultado obtido. Este novo resultado é o output da rede neural e deve ser comparado com o vetor da palavra "linguagem", já que é ela a palavra onde queremos prever dentro do contexto do exemplo dado. O erro então deve ser calculado e através da backpropagation [3], a rede neural vai ser atualizada com um novo peso W que foi calculado, fazendo com que os resultados obtidos no output tenham uma redução do erro a cada iteração do treinamento. No final do treinamento o W passa a ser o word mebedding gerado. Para o modelo do Skip-Gram a ideia é a mesma mas, o input é invertido com o output, ou seja Skip-Gram e CBOW são versões espelhadas um dos outros. Logo, no exemplo usado acima, teríamos agora a palavra "linguagem" como input enquanto as palavras "melhor" e "programação" estariam no output. Da mesma forma, como anteriormente, a matriz de peso W vai ter que ser ajustada através de backpropagation para diminuir o erro, só que agora é comparado com o erro das outras duas palavras, "melhor" e "programação".

Entre os dois modelos, o que melhor se aproxima do propósito do *Aplicai* é o Skip-Gram. Como discutido no paper do Mikolov (6), Skip-Gram é melhor para ser treinado com menos quantidade de dados, o que pode ajudar no processo inicial do aplicativo e além disso consegue representar melhor palavras menos frequentes, quando uma skill aparece poucas vezes no aplicativo esta será melhor representada quando comparado ao modelo do CBOW. Outro fator também é que no paper (3) utilizando as skills para treinar o seu modelo, eles tiveram melhores resultados utilizando o Skip-Gram.

[2] https://en.wikipedia.org/wiki/Activation$_{f}unction$
[3] https://en.wikipedia.org/wiki/Backpropagation

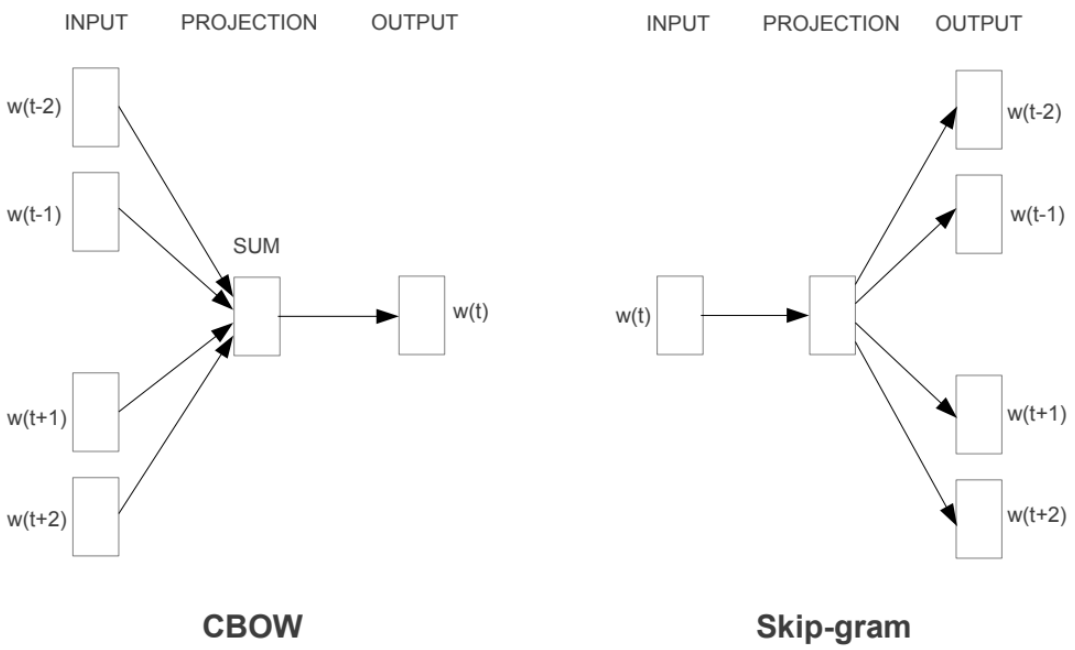

Figura 4.1: CBOW e Skip-Gram (6)

Baseado no trabalho do Skill2Vec (3), a ideia é de que skills podem ser usadas para gerar um word embedding que utilize a comparação de similaridades entre as skills como uma distância semântica. Podemos pensar que isso também pode valer para o contexto do *Aplicai*. Queremos construir um word embedding que entenda das relações semânticas entre as skills que fazem parte do universo do aplicativo. A diferença, agora, é que vamos usar apenas as skills como input. Vamos então supor com um exemplo de dez demandas e que em nove delas as skills Java, J2ee, Spring aparecem sempre juntas, enquanto Java e Javascript aparecem juntas apenas uma única vez. De forma análoga ao exemplo anterior queremos executar o processo do Word2Vec em cada uma das skills até obter vetores confiáveis. Uma vez que essas skills foram treinadas a skill Java vai passar a ter um vetor próximo de J2ee e Spring, o mesmo vale para as demais. E como a skill Java aparece poucas vezes com as outras skills como Javascript, a recomendação será sempre mais tendenciosa para as skills que tem vetores mais próximos como J2ee ou Spring. Se tivermos um treinamento sendo processado de forma constante, tanto com as demandas existentes quanto com as novas que serão ainda cadastradas, vai ser possível manter o word embedding sempre atualizado.

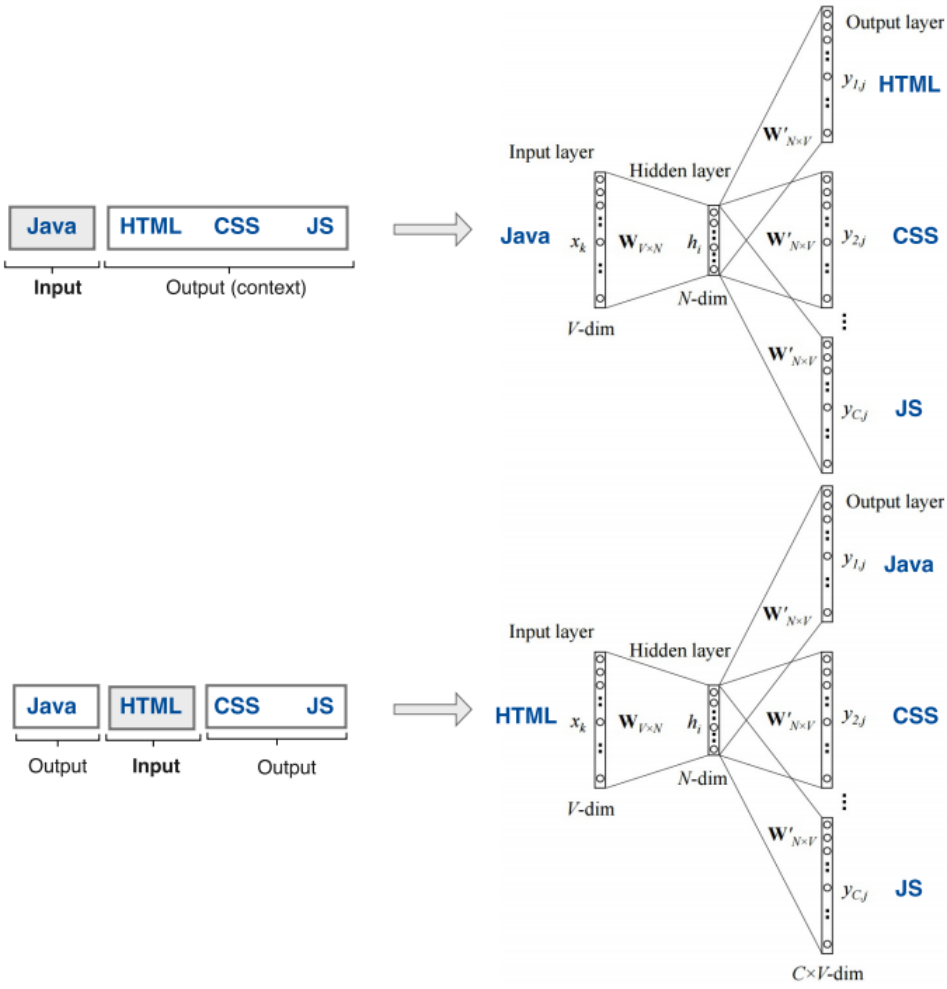

Figura 4.2: Skip-Gram processando com skills (3)

4.2 Método Word Mover Distance

Através de um word embedding conseguimos calcular as similaridades entre os vetores, usando técnicas como cosine similarity, distância euclidiana, etc. Word Mover Distance (8) é uma das técnicas para que seja possível calcular a distância entre documentos. Documentos no caso, podem ser entendidos como um conjunto de palavras, onde cada palavra é um vetor. Sabemos que os vetores gerados pelo Word2Vec tendem a preservar o relacionamento semântico das palavras. Com o WMD (Word Mover Distance) de forma natural conseguimos incorporar o conhecimento gerado pelo Word2Vec. Dessa forma o WMD sugere que podemos calcular o grau de proximidade semântica entre palavras. Se temos dois documentos, em que cada um tem um conjunto de palavras, é feito o cálculo da distância mínima em que cada palavra de um

documento leva para viajar até outro documento. Podemos então, pensar no seguinte exemplo contido no paper (7) com as seguintes frases:

- D0 = "Obama speaks to the media in ilinois"
- D1 = "The president greets the press in Chicago"
- D2 = "The band gave a concert in Japan".

O primeiro passo é eliminar as stopwords. No próximo passo devemos representar as frases usando bag of words[4] normalizada. Ou seja, nos documentos D0, D1 e D2 vamos ter apenas quatro palavras no vetor uma vez que as stopwords foram removidas, gerando a seguinte bag of words para os três documentos, V0, V1, V2 = [0.25, 0.25, 0.25, 0.25]. Vamos supor que queremos saber qual é o documento que seria mais similar ao D0. Usando a distância euclidiana deve-se achar agora os vetores com menor distância entre cada palavra. Vamos supor então que Obama em D1 tem a menor distância da palavra presidente, então devemos multiplicar 0.25 com a distância calculada entre Obama e presidente. Esse processo deve ser realizado entre cada palavra do documento de D0 e D1, e o mesmo processo entre D0 e D2. No final ao somar a distância total, vamos então saber que o documento que possui como resultado a menor distância, é considerado o mais similar. A imagem 4.3 representa bem o exemplo discutido acima.

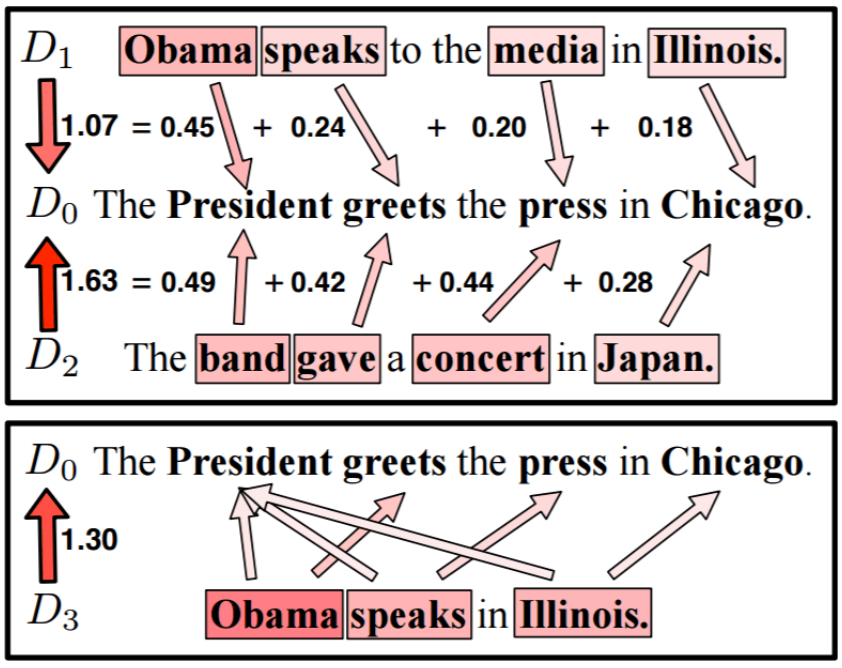

Figura 4.3: Exemplo de cálculo de distância entre documentos (7)

[4]https://pt.wikipedia.org/wiki/Modelo$_saco - de - palavras$

Trazendo para o contexto do aplicativo, vamos ter no lugar dessas frases um conjunto de skills. Vamos supor então que no lugar temos os seguintes documentos: D0 = [HTML, CSS, PHP] , D1 = [Java, Spring, J2ee] e D2 = [Javascript]. Da mesma forma como foi calculado no exemplo anterior, podemos então saber qual é a distância entre D0 e D1 e entre D0 e D2, utilizando apenas skills como base. Dependendo do word embedding gerado, o WMD pode indicar que entre D2 e D0 e entre D1 e D0, qual tem a menor distância é o D2, já que no geral Javascript tem uma valor semântico mais próximo de HTML, CSS e PHP uma vez que são tecnologias mais voltadas para o frontend. Inicialmente não é possível estabelecer para uma máquina uma relação de que Javasrcript é uma skill que está próxima semanticamente de HTML, CSS ou PHP. Isso vai ser apenas possível quando o aplicativo passar a ter quantidade suficientemente grande de demandas onde essas skills aparecem juntas, criando essa relação semântica durante o treinamento do Word2Vec. Portanto, no momento em que for criado o word embedding baseado nas skills, vai ser possível usar o WMD para ranquear as demandas na ordem decrescente das distâncias encontradas por cada documento.

5
Metodologia

5.1
Linguagens e ferramentas utilizadas

Para o desenvolvimento do sistema, foram utilizadas três linguagens de programação: Dart, Python e Javascript. Como para a construção das aplicações mobile/Web foi utilizado Flutter [1], obrigatoriamente a linguagem de desenvolvimento deve ser o Dart[2], pois é ele que interage com o SDK. Quanto ao desenvolvimento do sistema de recomendação, foi utilizado em sua grande parte a biblioteca Gensim [3] que usa a linguagem Python e fornece métodos para processamento de linguagem natural. Também foi utilizado Docker [4] para facilitar a execução da API de recomendação com Flask. Por último, funções serverless [5] do Cloud Functions, dispõe apenas do backend com Node.js, logo a linguagem utilizada deve ser o Javascript. Todo o projeto foi desenvolvido sendo versionado pelo git, de forma que nenhum código fosse perdido.

De forma a possibilitar testes para o aplicativo, foi utilizado o Android Emulator. Este foi criado usando a IDE Android Studio [6], o qual serviu apenas para este propósito. Para o desenvolvimento do código foi utilizado a IDE do Visual Code Studio [7] da Microsoft, usando plugins do próprio Flutter que auxiliam na testagem do aplicativo. Para realizar testes para os métodos do Word2Vec e o Word Mover Distance foi utilizado o Jupyter Notebook [8], que é uma ferramenta para Python que ajuda na visualização dos dados. O aplicativo também foi integrado com o Firebase [9] durante a fase de testes, de forma que fosse possível testar todas as iterações e possíveis falhas que poderiam ocorrer entre os sistemas.

[1] https://flutter.dev
[2] https://dart.dev/
[3] https://radimrehurek.com/gensim/
[4] https://www.docker.com/
[5] https://en.wikipedia.org/wiki/Serverless_*computing*
[6] https://developer.android.com/studio
[7] https://code.visualstudio.com
[8] https://jupyter.org/
[9] https://firebase.google.com

5.2 Desenvolvimento

Para dar início ao sistema, temos uma divisão de duas fases do desenvolvimento: uma para desenvolver o aplicativo frontend, por onde o usuário interage com a aplicação e a outra fase, o desenvolvimento de uma API de recomendação. Primeiramente foi feita uma integração do Firebase com o aplicativo para que toda a integração necessária fosse feita e para que durante o desenvolvimento já fosse iniciado com todos os recursos necessários do Firebase. Logo após, iniciamos o desenvolvimento do Aplicativo mobile, onde para cada tela uma interface básica era construída e caso tivesse que existir uma comunicação com um serviço externo, essa etapa vinha logo em seguida. Após vários testes, conforme uma tela ganhasse maturidade no desenvolvimento, uma melhoria na interface e correção de bugs também era feita posteriormente. Durantes essas etapas podemos citar a importância de utilizar o padrão BLOC, que ajudou na modularização da aplicação. A cada tela que era construída pelo aplicativo separava-se a parte de UI da lógica em um service layer. Conforme demonstrado na figura 5.1, é possível observar um caso de uso onde o BLOC ajuda a separar a lógica e a comunicação com agentes externos (por exemplo, Firestore) para reduzir a complexidade do código. Seu uso também é feito por mais de uma tela, evitando repetição de código. Uma vez que todas as telas já estavam prontas, realizei um teste completo para ver se todos os fluxos estavam de acordo com o esperado.

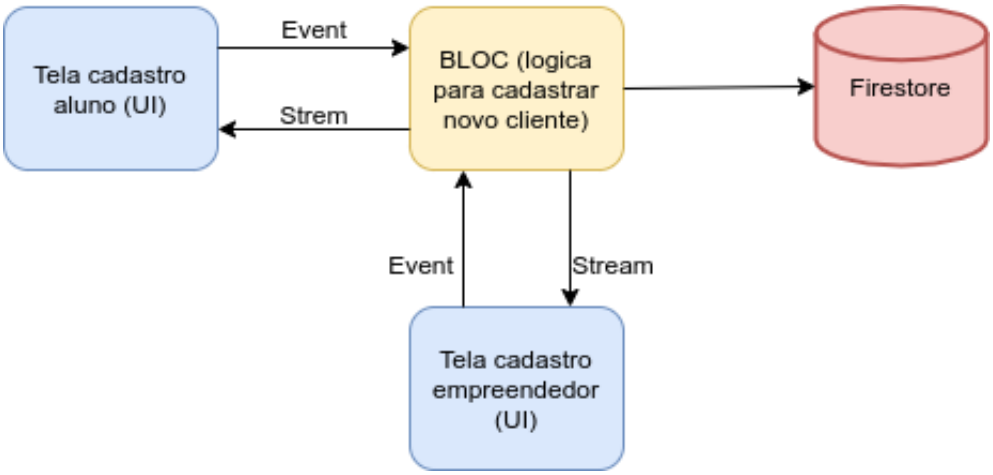

Figura 5.1: BLOC

Na segunda etapa iniciamos o desenvolvimento da API de recomendação. Mas antes mesmo de criar uma API, começamos a construção da lógica do algoritmo de recomendação pelo Jupyter Notebook para que o primeiro modelo de Word2Vec fosse testado e criado além de testar o Word Mover

Distance da biblioteca Gensim. Após a lógica já estar completa, utilizei o framework Flask para a criação da API REST. Dessa forma outros serviços conseguiam fazer requisições http e assim iniciar o processo de treinamento ou recomendação. Para que a API conseguisse ter acesso aos dados, também foi feita sua integração com os recursos necessários do Firebase. Com a API completa foi possível realizar todos os fluxos para todos os usuários desde o cadastro e login de usuário até receber uma recomendação de uma demanda.

5.3 Trabalhos relacionados

Diversos trabalhos buscam ajudar recrutadores a conseguirem fazer uma pré-avaliação de candidatos baseados em suas skills. Apesar de não ser um aplicativo para recrutadores, podemos seguir essa mesma fórmula, para que alunos consigam achar com mais facilidade e por conta própria uma demanda que seja mais apropriada ao seu perfil e preferências. Do mesmo modo, serve para que o empreendedor não tenha que correr atrás de participantes para suas demandas. Com isso a ideia do *Aplicai* se encaixa perfeitamente nesses trabalhos. Queremos uma forma de preferenciar os perfis de alunos que tenham skills relacionadas as demandas que vão ser recomendadas.

Hughes (8) demonstra que palavras que aparecem em um mesmo contexto tendem a ter um significado similar. Pensando nas skills do aplicativo, elas serão fornecidas pelos perfis de alunos ou demandas onde no momento em que for realizado um cadastro, estarão dentro de um mesmo contexto. O paper sobre o Skill2Vec (3), o qual serviu como base para a criação deste projeto, realiza um trabalho similar a proposta do *Aplicai*. Ele captura diversas skills utilizando web crawlers para montar uma base que possibilite criar um modelo de Word2Vec para fazer recomendações de skills. O trabalho realizado em (9) também é similar mas explora uma base do próprio Linkedin e utiliza o método LSA. O trabalho em (10) mostra que o método LSA tem uma performance pior que Word2Vec, quando o corpus disponível aumenta. Dessa forma, ao analisar o contexto do aplicativo, a ideia é que a sua base cresça continuamente, o que justifica a escolha do modelo Word2Vec. Em (3) parte deste trabalho foi disponibilizado no Github com a planilha que contém os dados das skills ofertadas. Por meio desta planilha, a ideia do sistema de recomendação foi possível ser colocada em prática, uma vez que foi possível avaliar o algoritmo construído em uma escala mais realística. Em (3) aponta que obteve resultados muito positivos. Realizando uma avaliação para seu método com um time de diversos peritos, demonstrou que 78% das skills que retornavam a partir de uma entrada, elas tinham uma relação muito próxima. Utilizando-se da planilha

disponibilizado no Github utilizado para o paper (3), foi possível criar um modelo Word2Vec inicial para o aplicativo e então iniciar avaliação a partir dele. Com este modelo conseguimos calcular a similaridade entre as palavras existentes dentro do modelo criado.

Um word embedding gerado a partir do Word2Vec consegue representar várias palavras e também consegue calcular as suas similaridades. Existem diversos métodos (11) que conseguem calcular a similaridade entre diferentes documentos. Kusner (7) demonstra que Word Mover Distance é um dos melhores métodos para realizar este cálculo usando um word embedding. Uma de suas vantagens é que não há necessidade de utilizar nenhum hyperparameter (12). Outra vantagem do método WMD, quando comparado a TF-IDF(13) é não precisar de ter exatamente as mesmas palavras nos mesmos contextos. O WMD não tem essa dependência, uma vez que ele se baseia na similaridade dos vetores. Vetores que estão próximos indicam que são palavras que estão contidas em um contexto próximo. Conforme representado pela figura 5.2, Kusner por meio de testes realizados com diferentes tipos de corpus, apresenta que WMD tem uma taxa de erro menor que a maioria dos outros métodos quando comparado. Utilizando-se do word embedding gerado pelo Word2Vec juntamente com a técnica do WMD, torna-se possível uma comparação de documentos.

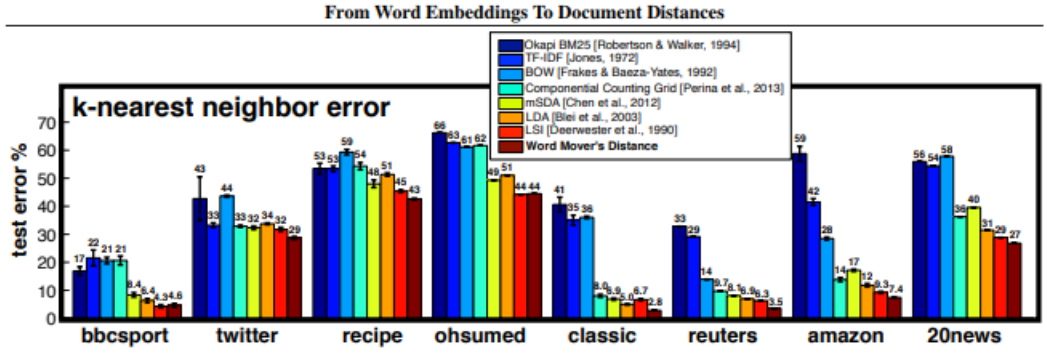

Figura 5.2: Gráfico comparativo entre diferentes modelos (7)

5.4 Cronograma

A figura 5.3 representa o cronograma de toda as etapas importantes para realizar a construção do *Aplicai*.

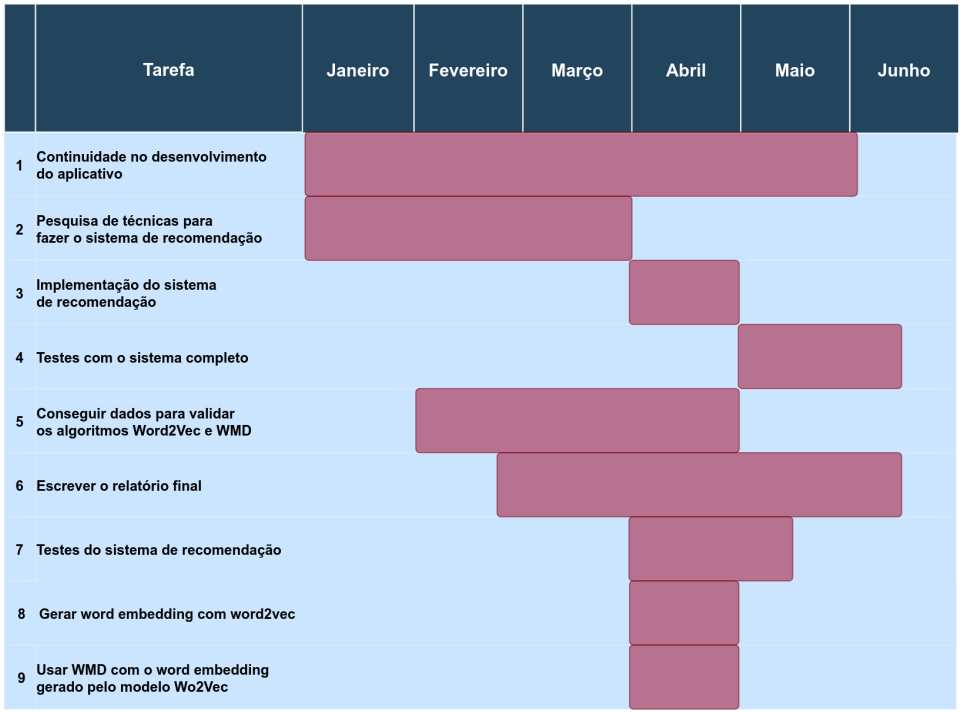

Figura 5.3: Cronograma

6
Implementação

Nesta seção será discutido a visão geral de toda a arquitetura do sistema, mostrando suas principais funcionalidades e fluxos. Como este projeto pode ser dividido em duas partes, uma parte dessa seção será utilizada a fim de explicar o funcionamento do aplicativo, demonstrando o desenvolvimento da interface, das funções, integração com o Firebase e como as telas do aplicativo interagem com o usuário. Na outra parte é discutido o desenvolvimento do sistema de recomendação, explicando o seu desenvolvimento e funcionamento com as bibliotecas utilizadas para construir a API que realiza ranqueamento das demandas.

O desenvolvimento do projeto utilizou a ferramenta Git para realizar o versionamento do código, além de salvar documentos e arquivos importantes para toda a fase da implementação. Para o desenvolvimento do aplicativo o código está disponível em `https://github.com/hugomachado93/aplicai`, enquanto o da API de recomendação está em `https://github.com/hugomachado93/aplicai-api`.

6.1
Implementação com o padrão BLOC

Para explicar as próximas etapas da implementação, é importante destacar que a parte lógica foi separada em três service layers, de modo que os métodos e funcionalidades da aplicação ficassem bem separadas de acordo com o seu objetivo. As três camadas são:

- AuthService - Camada onde ficam todos os métodos referentes a parte de autenticação:
- DemandService - Camada para tratar somente daquilo que for referente apenas a demanda
- UserService - Essa última camada é para métodos que lidam com o usuário

O padrão BLOC é usado em todas as telas existentes da aplicação. Logo nessa subseção, será dado enfoque sobre seu funcionamento utilizando um exemplo de implementação que foi feito para uma das telas do aplicativo.

Cada tela da aplicação tem seu próprio modulo BLOC, para que fique bem modularizado e não tenham muitos eventos e estados sendo controlados dentro de um mesmo módulo.

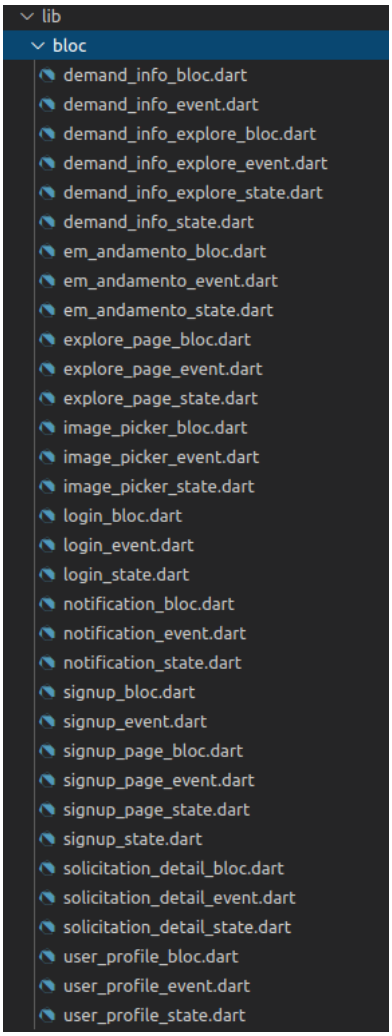

Figura 6.1: Relação de todos os BLOCs utilizados

Para que o padrão BLOC funcione, as três classes abaixo precisam existir: Uma classe de evento para enviar dados pela stream, a bloc para gerir o evento e a de state para fazer a mudança de estado da aplicação:

- DemandInfoExploreEvent
- DemandInfoExploreState
- DemandInfoExploreBloc

A classe DemandInfoExploreEvent é uma interface que deve ser implementada por todos os eventos que pertence ao BLOC do exemplo. Os eventos são responsáveis por iniciar um determinado fluxo no qual seus dados serão repassados para a classe DemandInfoExploreBloc por meio de uma stream. No

DemandInfoExploreBloc é onde está a camada que executa a lógica, que fica separado por uma outra camada de service. Comunicações com serviços externos com uma api ou com o banco devem ser nessa camada de service. A classe DemandInfoExplorePageState é uma interface que é usada para implementar os estados. Os estados dizem como a aplicação deve se comportar após receber um evento, trazendo os dados necessários na stream.

Nesse exemplo vamos supor que o evento emitido é o `GetCurrentUserAndEmployerData(String employerId)` que implementa a interface `DemandInfoExploreEvent`. No momento em que o evento é emitido deve ser passado o atributo employerId. Uma vez que o evento foi enviado pela stream, ele deve chegar na classe DemandInfoExploreBloc e então, dentro da classe DemandInfoExploreBloc é verificado qual evento foi acionado e então a lógica correta que deve ser executada. No caso o evento passado vai invocar o método `getCurrentUserAndEmployerData()` da classe UserService, que retorna um estado do tipo `DemandInfoExploreGetUserAndEmployerData` contendo os dados do usuário. A classe que vai escutar os eventos que no caso é a `DemandInfoExplorePage`, tem um método chamado de `build(BuildContext context)` que faz a renderização da UI. Nele é validado qual estado que foi passado. Então de acordo com o estado `DemandInfoExploreGetUserAndEmployerData` a aplicação exibe a interface correta utilizado dos dados que foram passados por stream pelo bloc.

A metodologia dos processos discutidos acima vai servir para qualquer tela do aplicativo. No geral cada tela que precisa acessar o banco de dados para buscar dados ou executar algum tipo de lógica vai seguir o mesmo procedimento, passa os dados necessários pelo evento, executa uma lógica e reage de acordo com os eventos e estados que são passados.

6.2 Cadastro e login (1 etapa)

A tela de cadastro inicial, realiza apenas o registro das informações básicas do usuário, seu e-mail e sua senha. Essa parte faz integração com o Firebase Authetication que é o responsável por armazenar os dados sensíveis como a senha. Para o aplicativo foram disponibilizadas duas formas de cadastro. Uma utilizando apenas o e-mail e uma senha da escolha do usuário e a outra usando uma conta existente do próprio Google. A classe que é utilizada para gerenciar toda a parte de autenticação do Firebase é a FirebaseAuth que contém todos os métodos necessários. Dentre eles os utilizados são:

- `Future<UserCredential> createUserWithEmailAndPassword(`
  `{`

```
    required String email,
    String password
});
```

- ```
  Future<UserCredential> signInWithCredential(
    AuthCredential credential
  );
  ```

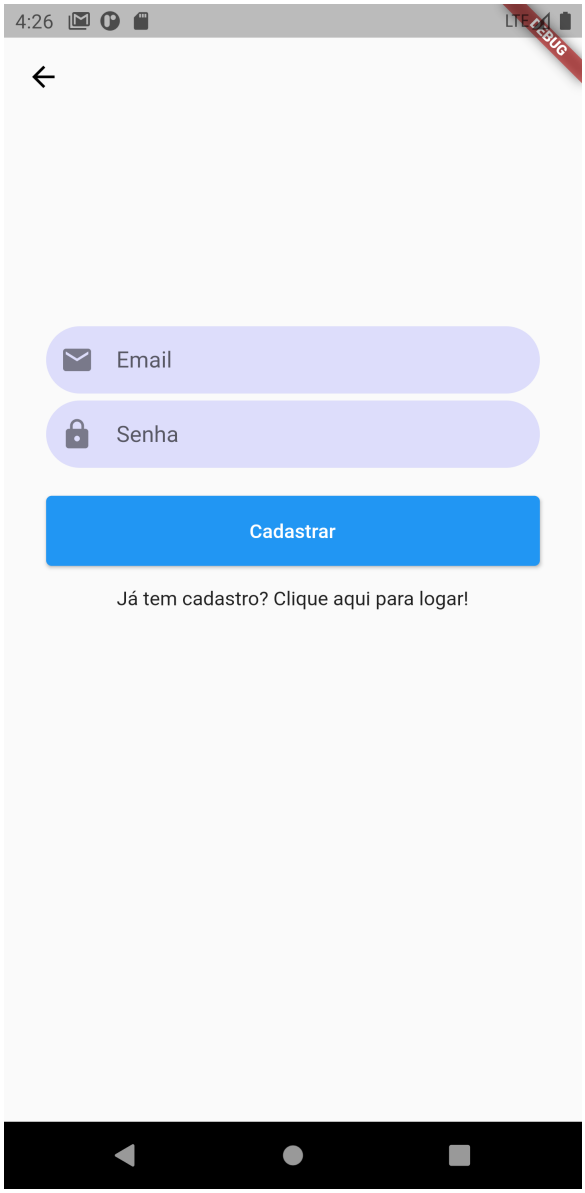

Figura 6.2: Tela de cadastro
  ```

6.2.1 Email and password authentication

A autenticação por e-mail e senha, acontece no momento em que um usuário está na tela de login, preenche os dados nos dois inputs da tela e clica no botão de cadastrar. O serviço do Authenticator é chamado através da função `createUserWithEmailAndPassword()` onde os dados são validados para verificar se o cadastro foi realizado com sucesso. Quando acontece qualquer erro, a função retorna a exception `FirebaseAuthException` com uma das quatro possíveis mensagens de erro:

- email-already-in-use
- invalid-email
- operation-not-allowed
- weak-password

No caso de qualquer exception gerada, o aplicativo não permite o cadastro do usuário e exibe para o usuário uma tela de erro de acordo com o tipo da mensagem de erro gerado pela exception. Caso contrário o usuário tem seu e-mail salvo e então é redirecionado para a próxima tela, onde deve dar continuidade ao seu cadastro.

6.2.2 Google Authentication

Se o usuário escolher autenticação pelo Google. Ela ocorre de uma maneira diferente. Neste caso o usuário tem que realizar um login com seu usuário do Google. `GoogleSignIn` é a classe que contém os métodos para realizar a autenticação pelo Google. O método `SignIn()` exibe a tela para o usuário logar. Após a autenticação esse mesmo método devolve tokens de acesso para que uma credencial seja criada. Essa credencial é usada pelo método `signInWithCredential()` que faz com que o usuário registre no Authenticator. Qualquer erro relacionado ao login e senha para este caso é gerenciado diretamente pela tela de login ou cadastro do Google, tirando a responsabilidade da aplicação de administrar essa etapa. Após o login realizado com sucesso conseguimos capturar alguns dados do usuário, e então seu e-mail será cadastrado no banco. Logo após este evento, este é redirecionado para a próxima tela para dar continuidade ao cadastro.

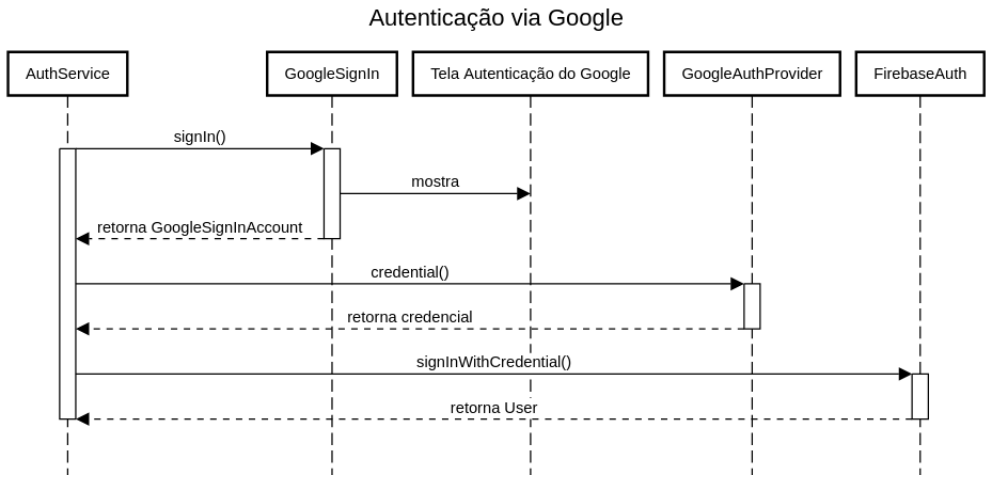

Figura 6.3: Autenticação via Google

6.3 Cadastro e login (2 etapa)

Uma vez que o usuário passou pela primeira etapa de cadastro, apenas é salvo no banco seu e-mail e também um atributo denominado `isFinished` com valor `false`. A senha é gerenciada pela próprio Authenticator, e não é salva no banco em nenhum momento. O atributo `isFinished` determina se um usuário já terminou de finalizar o cadastro. Caso contrário, se o usuário decide sair da aplicação e retornar depois, ao logar ele vai ser direcionado para a mesma etapa do cadastro de onde parou. Isso impede que o usuário consiga logar sem ter finalizado todas as etapas de cadastro. Uma vez na próxima tela, será exibido para o usuário qual tipo ele deseja ser, podendo escolher entre ser um empreendedor ou um aluno. Cada um apresenta formulários diferentes que devem ser preenchidos para que seja finalizado por completo. Uma vez finalizado, os dados do usuário são salvos no banco e o atributo `isFinished` aparece com valor `true`. Isso quer dizer que ao logar o usuário já está apto a usufruir de todas as funcionalidades da aplicação. O método `saveUserData()` da classe `UserService` é responsável por receber um objeto do tipo `UserEntity` ou `Empreendedor` e depois salvá-los no banco. Esse objeto tem todos os parâmetros que são preenchidos pelo formulário. Como são dois cadastros diferentes existe uma diferença entra seus atributos:

Tabela 6.1: Comparativo entre as classes UserEntity e Empreendedor

UserEntity	Empreendedor
bool isFinished	String companyName
String userId	String email
String name	String cnpj
String email	String razaoSocial
String cpf	String description
String curso	String linkedinUrl
String matricula	String portfolioUrl
String description	String urlImage
List categories	bool isFinished
String urlImage	`List<Demanda> demandas`
String linkedinUrl	
String portfolioUrl	
`List<Demanda> demandas`	

Uma vez que todo o processo de cadastro foi finalizado com sucesso, o usuário é direcionado para a tela de navegação principal. A figura 6.14 demonstra um diagrama de sequência que mostra cada passo do cadastro e login interagindo com as diferentes telas e serviços.

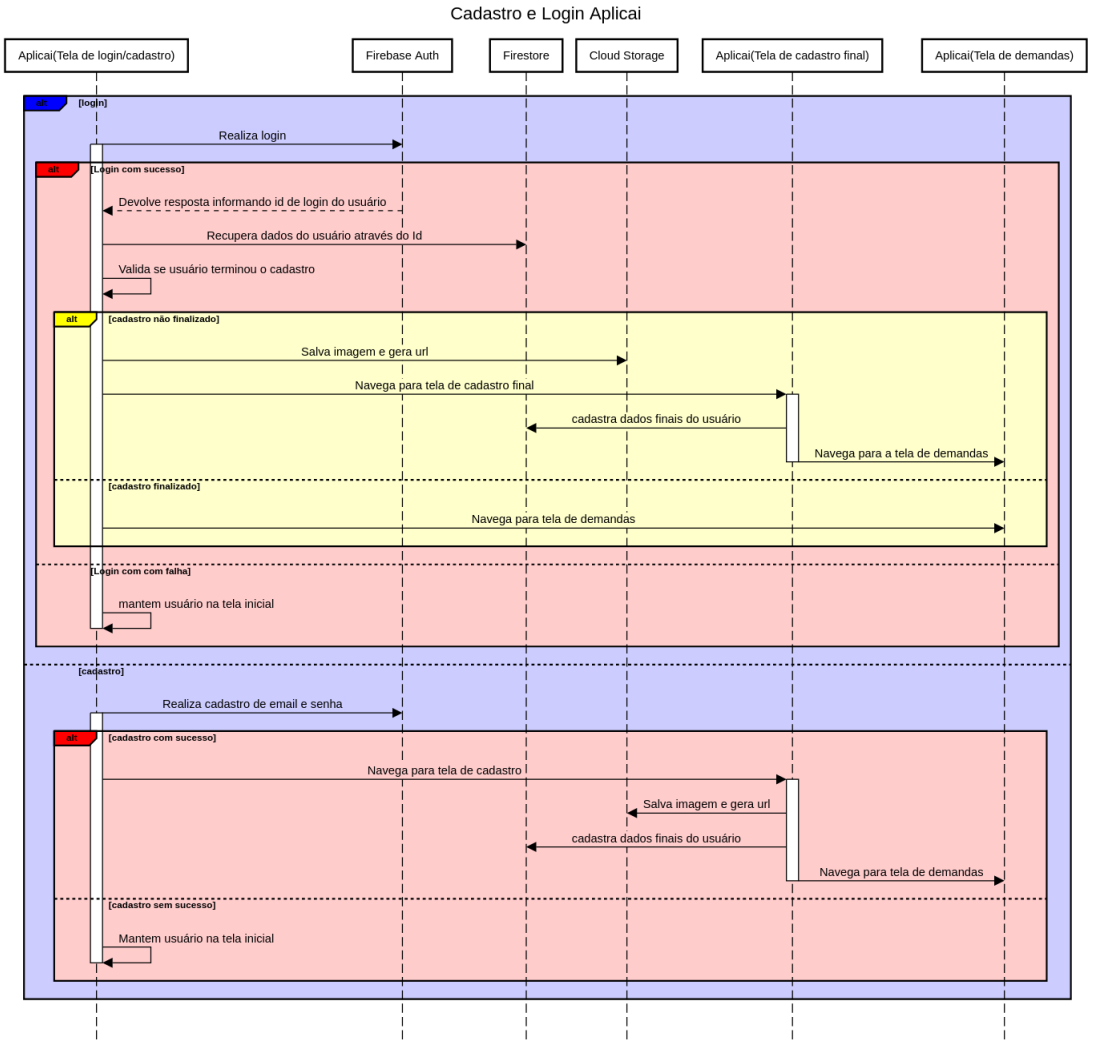

Figura 6.4: Cadastro e Login

6.4 Demandas

Nesta seção iremos tratar sobre como criar e participar de demandas, explicando detalhadamente passo a passo cada tela e as suas funcionalidades.

6.4.1 Cadastro de demandas

Usuários do tipo empreendedor têm a possibilidade de criar demandas e também de finalizá-las. Após seu primeiro login esta funcionalidade já está habilitada, basta apenas que o usuário navegue até a página de gerência de demandas. Na tela principal de navegação o usuário clica na aba “Em andamento”, onde ele será levado para uma tela que mostra as demandas no qual ele gerencia. Para criar uma nova demanda basta clicar no botão “+Nova demanda”. Um formulário com dados será exibido, no qual todos

os campos devem ser preenchidos. Para a criação da demanda o método `saveDemandData()` da classe `DemandService` é o responsável. O método deve receber um objeto do tipo Demanda. Abaixo é listado os seus atributos:

- String name;
- List categories;
- String description;
- Timestamp endDate;
- Timestamp startDate;
- String localization;
- String quantityParticipants;
- String parentId;
- String childId;
- String solicitationId;
- String urlImage;
- bool isFinished;
- double similarity;

Todos os atributos devem ser preenchidos por esta função, com exceção dos atributos `solicitationId` e do `similarity` que servem para etapas futuras(são usados pelo sistema de recomendação). Os atributos derivam através das informações do preenchimento do formulário. Assim que a demanda é criada, o usuário é direcionado de volta para a tela de gerenciamento das demandas. Toda demanda fica salva no banco e é exibida imediatamente para todos os usuários, que podem se cadastrar no mesmo momento. Para exibir, o atributo `isFinished` é preenchido com o valor `false`, indicando que usuários podem participar da demanda. Uma vez que foi finalizada seu atributo booleano troca para `true` e usuários não podem mais ingressar e além disso a demanda deixa de ser exibida.

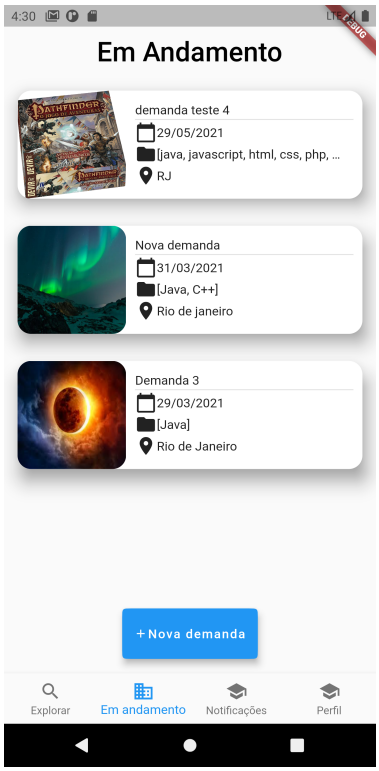

Figura 6.5: Tela de gerenciamento das demandas

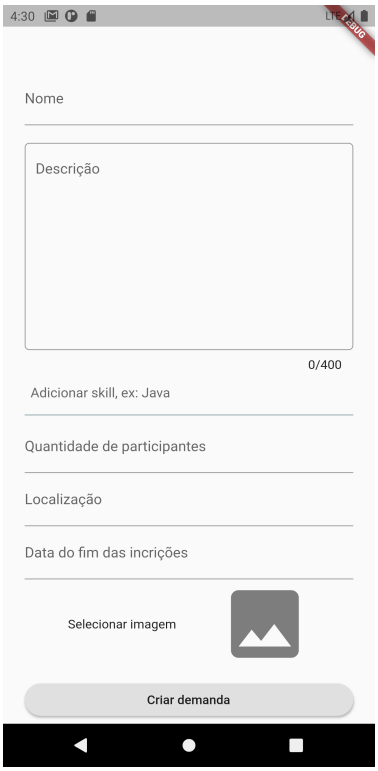

Figura 6.6: Formulário de criação da demanda

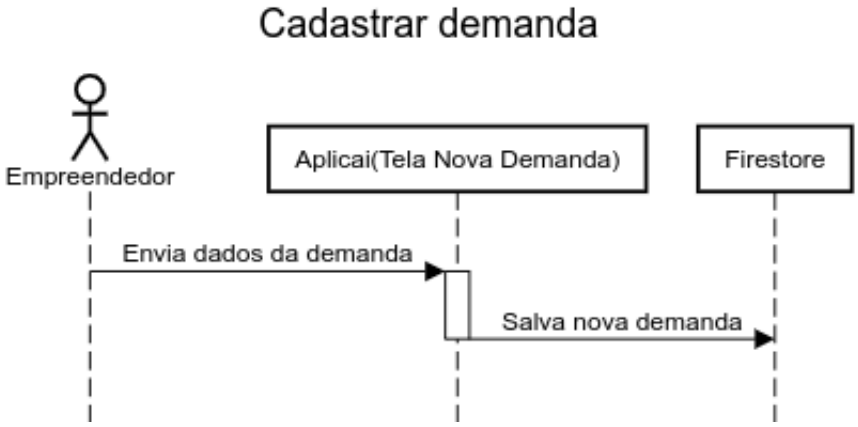

Figura 6.7: Diagrama de sequência sobre a realização do cadastro

6.4.2 Inscrição e escolha de uma demanda

Usuários do tipo estudante estão aptos a realizarem solicitações para participar de uma demanda. Usuários do tipo empreendedores não podem solicitar, mas podem visualizar informações de outras demandas que estão disponíveis. Na tela de navegação o usuário pode visualizar as demandas ativas

para cadastro clicando na aba “Explorar”. Dentro desta tela todas as demandas disponíveis são exibidas. Para realizar uma solicitação, basta clicar em uma das demandas e na próxima tela, será exibida algumas informações junto com um botão escrito “Quero me inscrever” que navega o usuário para a próxima tela. Uma caixa será exibida para que o usuário escreva um texto informando porque é um bom candidato para participar da demanda. Uma vez finalizado, basta clicar em “Quero me inscrever” e a solicitação será criada e uma notificação deverá ser enviada para o usuário dono da demanda informando que existe uma nova solicitação.

Quando o aluno solicitar, o método `createSolicitation()` será o responsável por criar a solicitação. O método deve receber três identificadores diferentes, o id do usuário que criou a demanda, o da demanda e o do usuário que executou a solicitação e por fim o texto motivacional. Com isso uma solicitação fica vinculada à demanda pelo id de usuário que realizou a solicitação.

A figura 6.12 demonstra como os documentos representam as demandas junto aos usuários e solicitações feitas. O documento Demands tem o Id do usuário que criou uma demanda. O documento DemandList tem uma lista de documentos com todas as informações da demanda. Após a solicitação da participação, um novo documento Solicitation é criado junto com o id do usuário solicitante e seu texto motivacional.

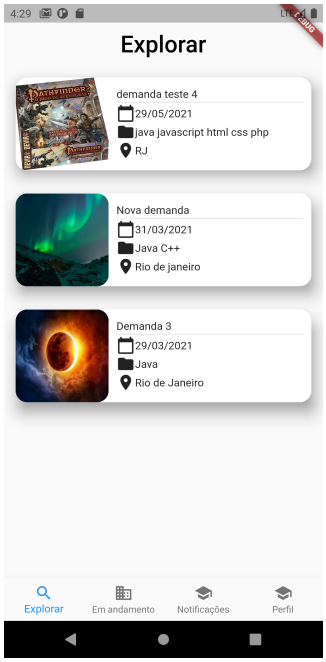

Figura 6.8: Tela de exploração das demandas

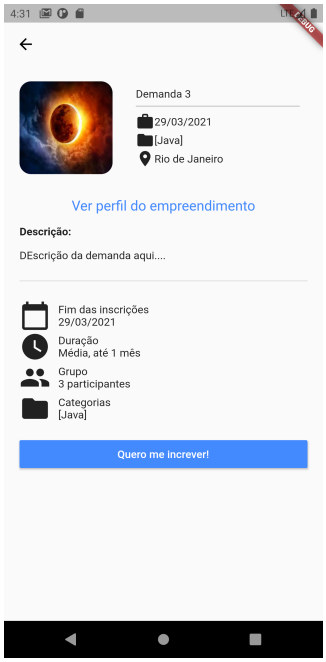

Figura 6.9: tela informação da demanda

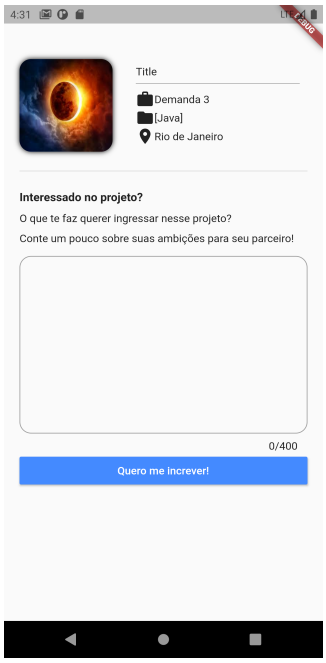

Figura 6.10: Tela para escrever uma solicitação

Figura 6.11: Tela com solicitação realizada com sucesso

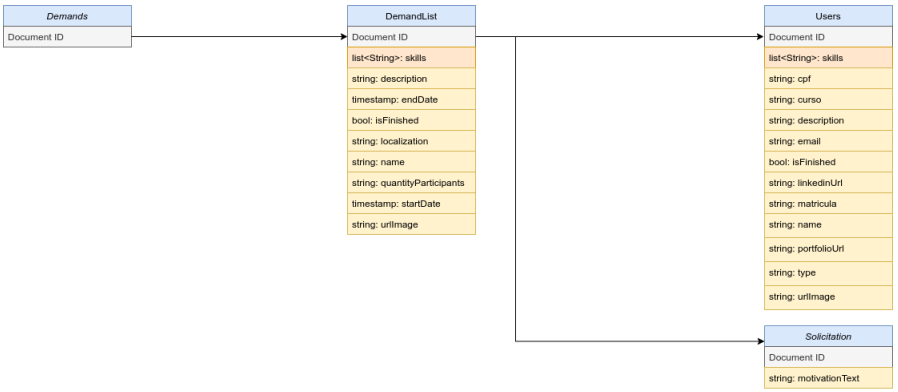

Figura 6.12: Modelo que representa as demandas, usuários e solicitações

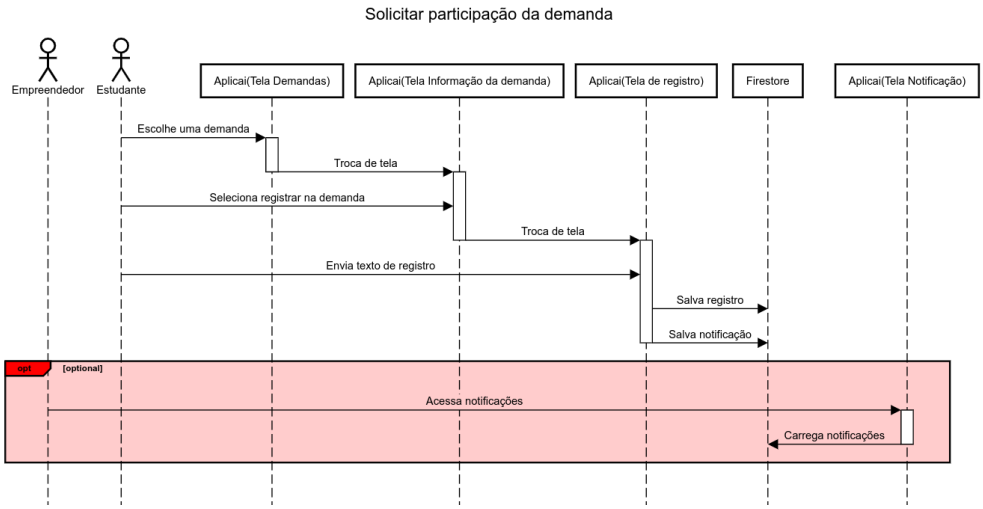

Figura 6.13: Diagrama de sequência sobre solicitar para entrar em uma demanda

6.4.3 Aceitar ou Recusar uma demanda

Após um usuário ter solicitado para fazer parte de uma demanda, o empreendedor recebe uma notificação avisando sobre o solicitante. Neste momento o empreendedor pode ir até a tela "Em andamento" e selecionar qual demanda ele deseja gerenciar. Após selecionado, ele deve clicar em "Ver solicitações", onde será direcionado para uma tela que mostra todos os usuários que realizaram uma solicitação. Uma vez que o empreendedor escolheu o aluno de interesse, será mostrado em uma outra tela o texto motivacional escrito pelo solicitante e também dois botões, um para aceitar e outro para recusar a participação deste candidato na demanda. Caso seja recusado a solicitação é removida e o usuário será notificado que não foi aceito. Caso contrário o usuário passa a fazer parte da demanda e também é notificado. O método `updateParticipantsOfDemand(Demanda demanda)` da classe `UserService` é utilizado para atualizar a lista de usuários que pertencem a uma demanda. De modo igual a demanda também passa a pertencer a um usuário. O motivo para ter dados duplicados é para simplificar as querys que são feitas para buscar os dados. O método `rejectUserSolicitation(Demanda demanda)` é chamado quando a solicitação é recusada e então ela é removida do usuário.

Uma vez que um usuário foi aceito, ele agora passa a fazer parte da demanda e consegue acessar pela aba "Em andamento". Ao clicar na demanda o aluno poderá ver algumas informações sobre ela e além disso é nessa mesma tela que fica disponível os dados do e-mail para contato, podendo o aluno então iniciar a primeira conversa com os integrantes.

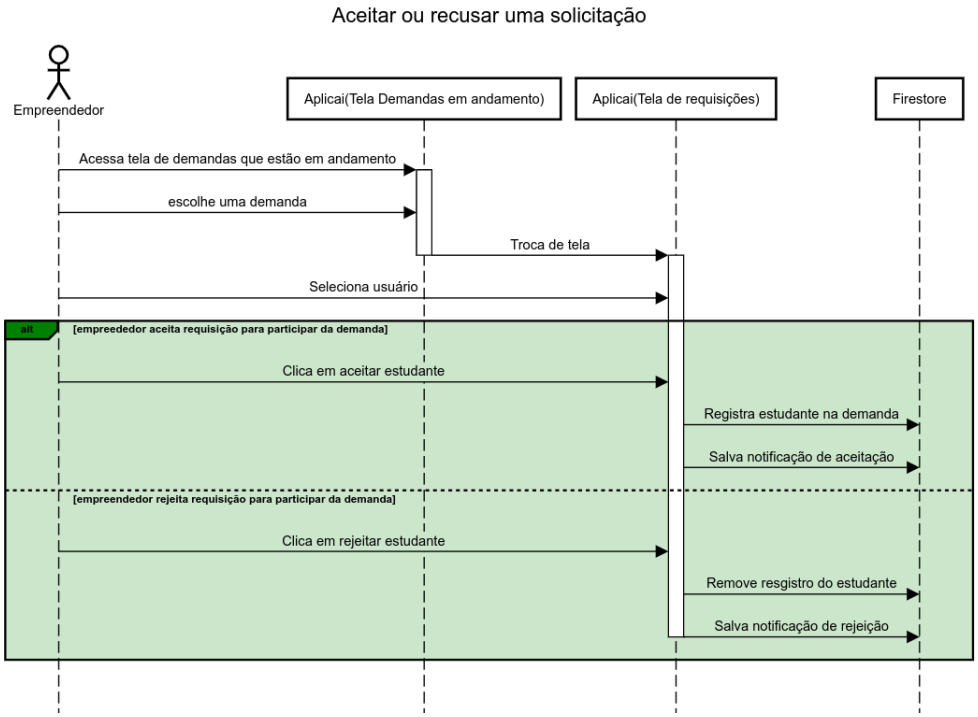

Figura 6.14: Diagrama de sequência sobre aceitar ou recusar uma solicitação

6.5 Notificações

Em diversas situações, notificações são importantes para que o usuário seja avisado de que alguma informação nova está disponível. A tela de notificação pode ser acessada pela tela principal de navegação. Nessa tela exibe uma lista de cartões onde cada um tem a descrição com as informações básicas para o usuário. As notificações são geradas salvando-as no banco de dados no documento Notifications. Cada notificação fica atrelada a um usuário. As notificações neste projeto são separadas por categorias, já que cada uma pode ter um formato diferente. O objetivo é criar uma régua de comunicação com o usuário, deixando-o sempre informado dos processos que acontecem no aplicativo, dessa forma garante um melhor engajamento do usuário com a aplicação. Abaixo são descritas os seguintes tipos de notificações que um usuário pode receber:

- Signup
    - Notificação quando o usuário realiza um cadastro
- Solicitation
    - Notificação quando uma solicitação é finalizada

- Request
  - Notificação quando uma solicitação é criada
- DemandRecomendNotification
  - Notificação quando o usuário tem novas recomendação de demandas disponíveis
- FirstRecommendation
  - Notificação quando o usuário já tem recomendações de demandas disponíveis pela primeira vez

Para que cada notificação tivesse seu formato ou layout diferente, um padrão do tipo strategy [1] é utilizado. A classe abstrata `NotificationsStrategy` é implementada por cada uma das notificação. O método `createCard(Notify notify)` deve ser então implementado e fica responsável por receber o objeto Notify, que tem todos os atributos necessários para renderizar a notificação. O método `invokeNotificationByType(Notify notify)` é responsável por ler o tipo da notificação e retornar o objeto de Notificação correto. Dessa forma a notificação pode ser renderizada de acordo com o formato e layout necessários de forma dinâmica. A notificação é criada por meio de um novo documento no banco com os dados da notificação. Essas notificações devem estar atreladas aos usuários, onde cada um tem suas próprias notificações. Abaixo um exemplo que insere uma notificação no banco:

```
      _db
      .collection("Users")
      .doc(demanda.parentId)
      .collection("Notifications")
      .doc()
      .set({
    "notification":
        "Você recebeu uma proposta de ${userEntity.name}
        para participar do projeto ${demanda.name}",
    "type": "request"
  });
```

Para recuperar uma notificação o método `getUserNotifications(UserEntity userEntity)` deve receber um parâmetro com o Id do usuário que através

[1] https://pt.wikipedia.org/wiki/Strategy

dele consegue buscar todas as notificações associadas e então disponibiliza-las para a visualização na aba "Notificações".

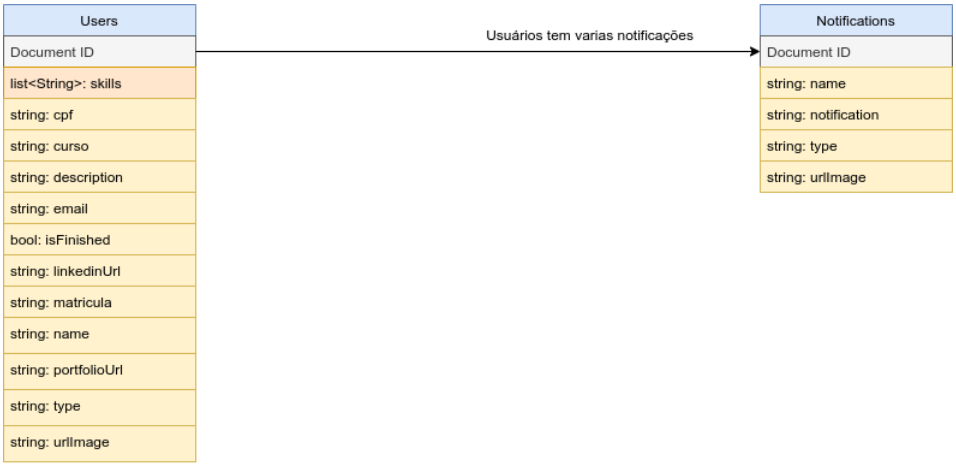

Figura 6.15: Diagrama notificações

6.6 Perfil do usuário

Os usuários da aplicação tem a possibilidade de visitar seu próprio perfil, assim como outros usuários também. No seu perfil ele consta algumas das informações que inseriu no momento que finalizou o cadastro na aplicação, além de mostrar as demandas do qual ele já participou previamente e já foram finalizadas. Como estudantes e empreendedores têm atributos diferentes na sua pagina de perfil, vai existir uma pequena variação quanto aos dados que são apresentados na tela. Uma das funcionalidades principais é que um usuário do tipo aluno consegue visitar o perfil de um empreendedor por meio das demandas que constam na aba principal. O objetivo é que os alunos possam entender melhor sobre quem é o empreendedor.

São duas das funções que carregam os dados do perfil do usuário `getUserStudentAndFinishedDemands()` e `getUserEmpolyerAndFinishedDemands()`, ambos os métodos vão carregar as informações dos usuários cadastradas no banco. As informações são sobre os dados do usuário assim como as demandas das quais ele participou. Com esses dados então a aplicação pode disponibilizar no perfil as informações necessárias. Dessa forma o próprio usuário pode observar melhor suas informações e além disso os outros usuários que visitarem seu perfil vão te uma ideia melhor sobre o tipo de demanda que ele tem interesse.

Figura 6.16: Tela do perfil do empreendedor

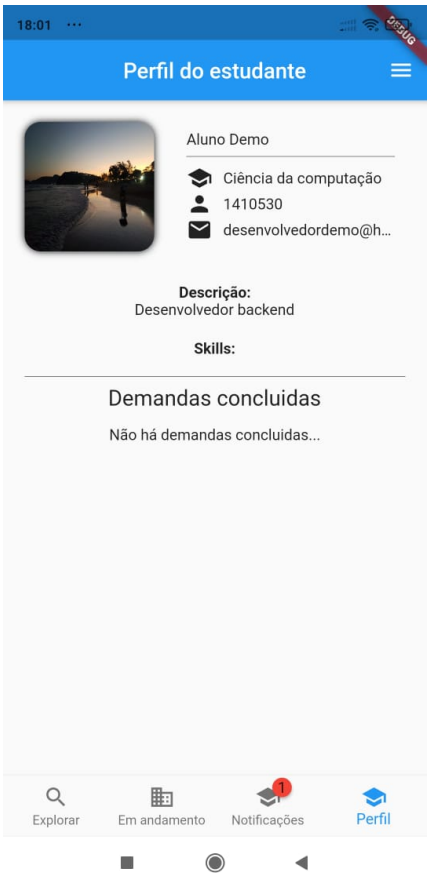

Figura 6.17: Tela do perfil do estudante

6.7 Sistema de Recomendação

Nesta seção será relatado como foi o desenvolvimento da API de recomendação juntamente com o algoritmo que teve que ser desenvolvido. A ideia do sistema de recomendação é que ele seja separado do aplicativo, para que facilite o desenvolvimento e a integração com diferentes aplicações. Para isso foi desenvolvido uma API usando o framework Flask, que permite criar endpoints [2] para que outras aplicações consigam realizar comunicações entre si. O sistema deve funcionar de forma assíncrona ou seja os usuários do aplicativo não vão receber nenhuma recomendação de demandas logo quando se cadastram. O processo de recomendação deve ser realizado de tempos em tempos, onde o usuário é notificado quando já tem novas recomendações disponíveis. A API pode ser dividida em duas partes, uma que será responsável pelo treinamento, onde vai ser gerado o primeiro word embedding e dará continuidade

[2] https://en.wikipedia.org/wiki/Communication_endpoint

a seu treinamento, e a outra onde será feito de fato o processamento para ranquear as demandas. São dois endpoints diferentes presentes na API:

- GET /trainning
- GET /async-recommendation

Os endpoints são responsáveis por iniciar o processo de execução do treinamento e da recomendação respectivamente. Usar o Word2Vec e o WMD pode ser um processo longo e levar horas para finalizar dependendo do tamanho da base que será usada. Por este motivo essas etapas estão separadas. O objetivo é que os processos sejam executados por uma crontab[3]https://pt.wikipedia.org/wiki/Crontab, que agendará a execução dos métodos de treinamento e recomendação através de uma requisição http para os endpoints citados acima.

6.7.1 Treinamento

O primeiro passo antes mesmo de recomendar é gerar um Word Embedding pelo treinamento das skills encontradas nas demandas. Para iniciar o processamento do treino deve ser feita uma requisição http para o endpoint `/trainning` que é acionado por uma crontab do sistema. A ideia é que exista um agendamento com horários específicos para realizar o trenamento. A biblioteca Gensim que é utilizada para construir o modelo Word2Vec tem a possibilidade de criar um modelo próprio onde este carrega informações do Word Embedding assim como configurações específicas do framekwork. O método `Word2Vec(sentences=corpus, min_count=2, vector_size=300, window=5, sg=1)` do Gensim é utilizado para gerar o modelo pelo treinamento do corpus. Abaixo é demonstrado o significado de cada parâmetro:

- sentences - Parâmetro que recebe o corpus para usar no treinamento.
- min_count - É a frequência de vezes que uma palavra deve aparecer para ser considerada no processamento.
- vector_size - É a dimensão do Word Embedding gerado
- window - É o número de palavras a serem consideradas ao redor do input
- sg - Valor boolean que diz se é para usar o modelo Skip-Gram ou CBOW

[3]

Antes de iniciar, usando a base do Firestore, deve ser filtrado de todas as demandas o vetor de skills dela, independentemente se ela foi finalizada ou não. Quando todas as skills forem capturadas, já temos dados suficientes para construir o corpus necessário para seguir com o treinamento. O corpus gerado e as configurações necessárias são passadas no método `Word2Vec()`. Vale destacar que o tamanho do `window` é o maior possível, pois vamos considerar que todas as skills dentro de uma demanda são importantes. O valor de `sg` também será o Skip-Gram pois como discutido anteriormente ele consegue representar melhor palavras menos frequentes. Uma vez que o treinamento foi finalizado o Gensim tem a possibilidade de salvar o word dembedding gerado dentro de um arquivo. Esse arquivo é sempre salvo no Cloud Storage com a finalidade de nunca perder o word embedding e também de poder continuar treinando para os próximos agendamentos, dessa forma vai estar sempre em constante evolução e sempre que aparecer novas demandas e até mesmo novas skills estas serão introduzidas para seu vocabulário. Dessa forma conseguimos garantir a construção de um grande dicionário que sempre estará a par das novas tecnologias.

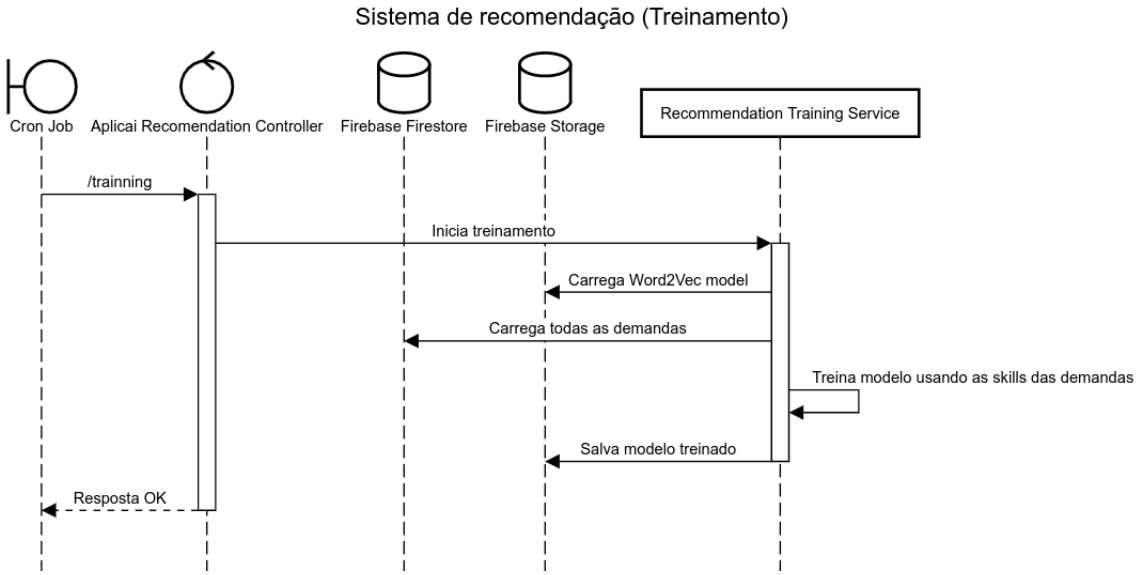

Figura 6.18: Treinamento

6.7.2 Recomendação assíncrona

O processamento de recomendação das demandas pode ser inicializado a partir do momento que existe um modelo de word embedding salvo no Firebase Storage. O último modelo salvo no Storage é o que vai ser utilizado para reco-

mendar. A técnica que vai ser utilizada para recomendar é o WMD que foi discutido em seções anteriores. O Gensim possui o método do WMD que é disponibilizado pelo função `WmdSimilarity(corpus, w2v_model, num_best=100)`. Abaixo é demonstrado o significado de cada parâmetro:

- corpus - Recebe uma lista de skills no parâmetro
- w2v_model - Recebe um modelo de word embedding
- num_best - Número dos n melhores documentos que devem ser ranqueados

O início do processamento se dá no momento em que a crontab faz uma requisição http para o endpoint /async-recommendation. No primeiro passo o último modelo de word embedding é carregado do Firebase Storage. Nesse momento queremos filtrar todas as skills do perfil do usuário além das skills presentes nas demandas do qual ele participou. Dessa forma conseguimos expandir a recomendação para além do número de skills que consta apenas em seu perfil. Agora é a vez de coletar as skills de todas as demandas existentes no projeto, pois a ideia é descobrir qual delas que devem ser recomendadas para o usuário.

Tendo agora todos os dados necessários, utilizando a função `WmdSimilarity()`, deve se passar como corpus todas as skills das demandas. Para o modelo deve ser o word embedding gerado pelo último treinamento anterior e para o num_best o número das n melhores demandas a serem recomendadas. O parâmetro num_best deve ser passado como query string na url da requisição http, para que fique facilmente parametrizável possibilitando a escolha da quantidade de recomendações e caso não seja passado nenhum valor, será utilizado o valor 100 que é o padrão. A função `WmdSimilarity()`, deve retornar uma instância por onde deve ser passado uma query de busca e como resultado retorna o índice e valor da similaridade dos n melhores documentos. Na query deve ser passada todas as skills pertencentes ao perfil do usuário e das demandas do qual ele participou. Com o índice das melhores demandas, já existe a possibilidade de fazer uma recomendação para o usuário mas esta não é viável ser feita de forma síncrona, ou seja, o usuário não deve ficar esperando na tela de exploração de demandas o processamento do WMD, caso contrário o usuário teria que aguardar muito tempo até fossem exibidas as demandas. Isso é devido a alta complexidade de $O(p^3 logp)$ do WMD onde p é o número de palavras únicas. Com um grande volume de skills que é o caso do *Aplicai* isso afetaria diretamente o tempo de resposta da aplicação com o usuário.

Para contrapor este problema, a solução foi realizar uma recomendação assíncrona utilizando o banco do Firestore. Como já temos os índices e o valor da similaridade das demandas ranqueadas que foram geradas pela função `WmdSimilarity()`, devemos então criar um documento no banco apenas com estes dados para cada usuário da aplicação. O documento a ser criado é denominado Recommendation e deve ter como ID o identificador do usuário de quando ele se cadastrou no aplicativo. Como dentro do banco já existe os documentos com as informações específicas de cada demanda, apenas os IDs e dados cruciais devem ser salvos no documento Recommendation, para que posteriormente o aplicativo consiga recomendar as demandas de forma a evitar muita duplicidade de informações. Nesse mesmo documento deve ser criada uma lista que contenha as informações como: o id da demanda, o seu grau de similaridade e o id do usuário empreendedor que criou a demanda. Através das informações desse documento vai possibilitar que o aplicativo consiga buscar as demandas e disponibilizá-las para o usuário ver as suas recomendações. Agora pelo aplicativo a classe `DemandService` possui o método `Future<List<Demanda» getActiveDemands()` responsável por buscar as demandas contidas no documento Recomendation e deve mostrar na ordem de acordo com a sua similaridade. Para isso o documento tem o campo `similarity` que indica o grau de similaridade das skills do usuário com a demanda. O método também deve lidar com os casos dos usuários que acabaram de se cadastrar e ainda não tiveram sua recomendação de demandas processadas. Para este caso, as demandas devem estar disponíveis mas vão aparecer de forma aleatória em sua tela.

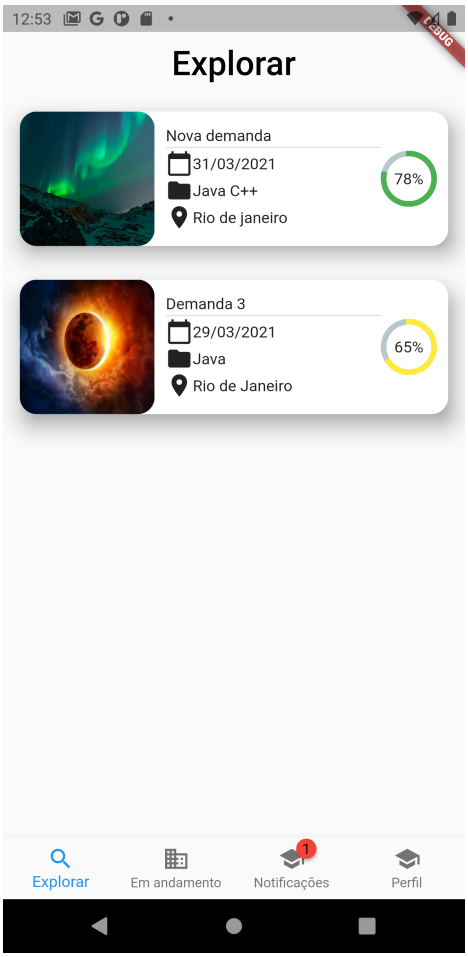

Figura 6.19: Tela com demandas recomendadas

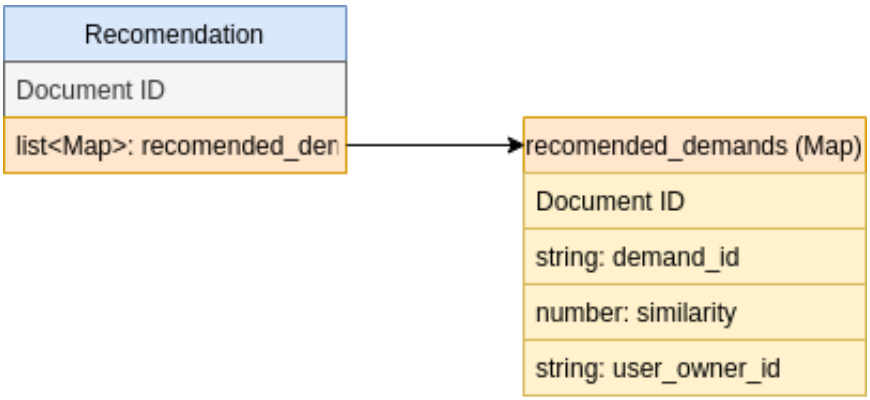

Figura 6.20: Diagrama do documento Recommendation

Para que o usuário seja notificado quando tem suas primeiras ou novas demandas recomendadas, a aplicação utiliza os eventos do Cloud Functions, que ocorrem por meio de atualizações ou inserções no banco Firestore. Isso faz com que a notificação seja imediata e não precisemos realizar o trabalho

de ficar observando uma alteração no documento Recommendation. Para este projeto foram criadas duas funções:

- `functions`
  `.firestore`
  `.document('Recommendation/{userId}')`
  `.onUpdate(async (change, context)`
- `functions`
  `.firestore`
  `.document('Recommendation/{userId}')`
  `.onCreate(async (snap, context)`

A função `onUpdate(async (change, context)` deve ser responsável por notificar ao usuário que já existem demandas recomendadas, mas este deve ter uma atualização na sua lista de recomendações. O Método recebe as informações das demandas antes do update e após o update no documento. Desse modo é possível verificar se teve qualquer modificação nos itens recomendados, caso tenha, uma notificação deve ser enviada ao usuário comunicando quantas foram atualizas na sua lista de recomendação. Já a função `onCreate(async (snap, context)` é responsável para quando o usuário nunca teve uma demanda recomendada. Nesta função não há necessidade de qualquer validação pois, caso seja acionada, já sabemos que esse usuário nunca teve recomendações. Basta apenas enviar uma notificação ao usuário avisando que ele agora já pode ver quais são suas recomendações.

O objetivo dessas notificações é deixar claro ao usuário quando, de fato, ele tem recomendações. Sendo assim, uma boa forma de mantê-lo inteirado sobre o que está acontecendo.

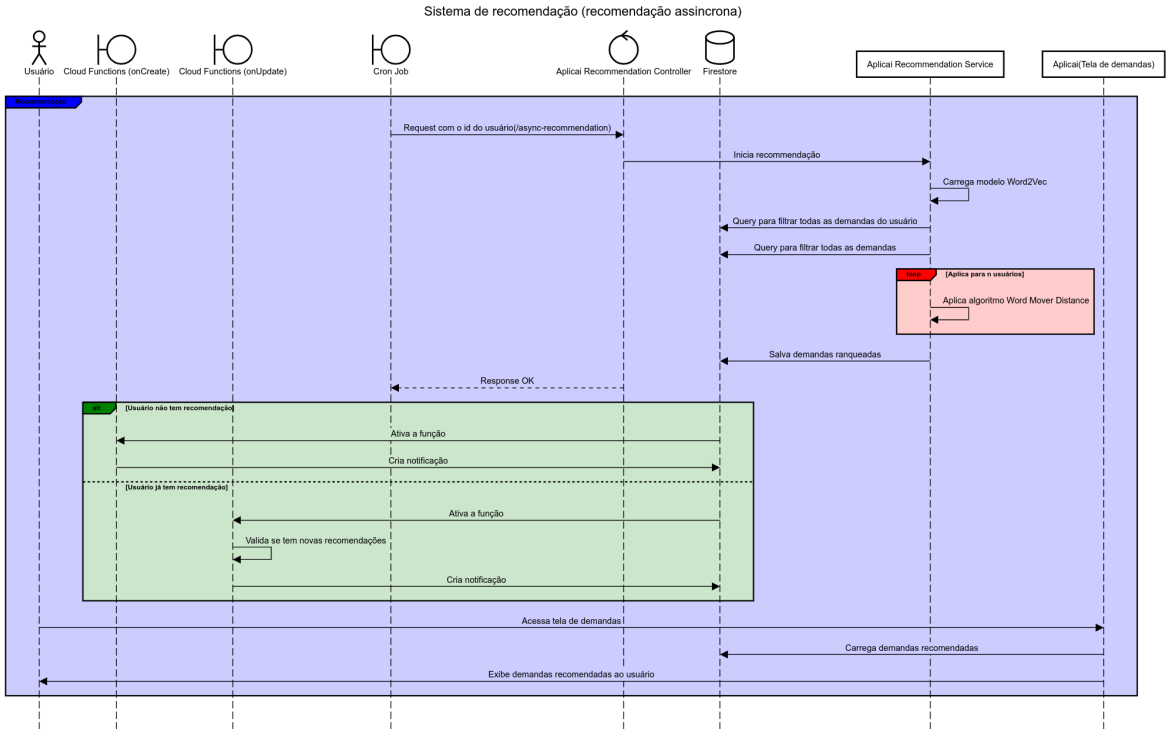

Figura 6.21: Recomendação assíncrona

7
Testes de avaliação

O objetivo dessa seção é demonstrar os testes que foram realizados para validar o sistema de recomendação junto com a aplicação. Para validar que a ideia do sistema de recomendação utilizando o Word2Vec em conjunto com o WMD é funcional na prática, é importante utilizar um grande volume de dados reais. O paper (3) foi a base teórica para este projeto e disponibilizou no Github arquivos em formato csv que contém skills do qual foram retiradas direto de sites de oferta de emprego. Para validar os testes foi utilizado o Jupyter Notebook como ferramenta para visualizar e testar a criação do word embedding.

O csv utilizado contém 50 mil ofertas de empregos onde cada uma pode ter dezenas de skills. As skills podem ser de diversas áreas e não contemplam apenas a área da computação. Como o foco desse projeto são skills voltadas para esta área, foi fundamental que tivessem números suficientes para colocar em prática. Uma das vantagens de se utilizar as skills tecnológicas é que são globais e independentes da língua nativa. Analisando as skills que mais aparecem, é possível determinar que entre as mais populares são as da área tecnológica. Podemos ver na figura 7.3 que Java, Javascrit, SQL, etc... aparecem logo como as mais populares.

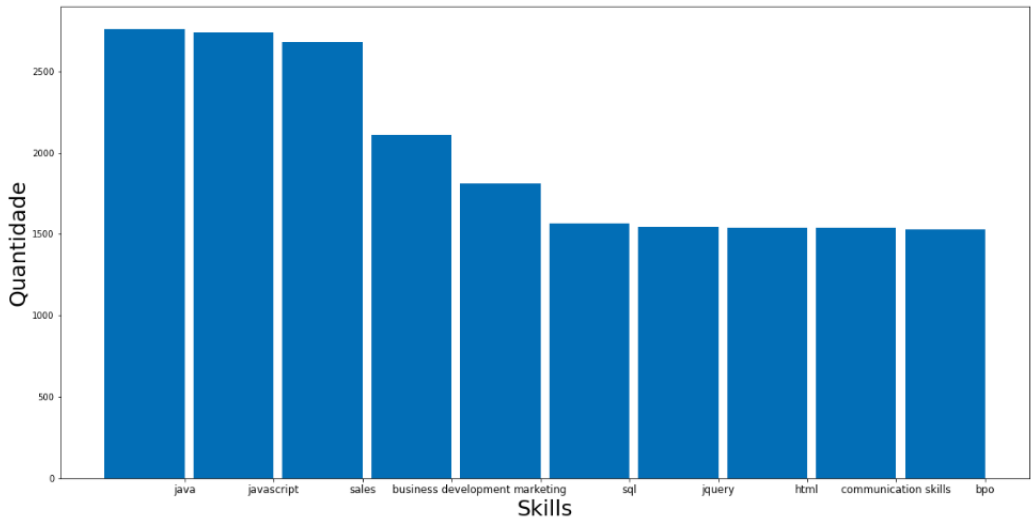

Figura 7.1: Skills que mais aparecem no csv

Como os dados apresentam uma quantidade suficiente para iniciar a criação do word embedding, a partir dessa base vamos então criar o modelo Word2Vec com o Gensim e gerar um word embedding, a partir das skills do csv. Para este teste é importante destacar alguns dos parâmetros que foram utilizados para a criação do modelo. O parâmetro `window` será o maior possível pois será considerado que todas as skills presentes são importantes e o modelo utilizado é o Skip-Gram.

Uma vez gerado, agora vale visualizar se as skills de fato tem uma relação entre elas. Para isso foi utilizado o t-SNE que possibilita o plot de vetores em várias dimensões trazendo a visualização para uma dimensão 2D. Para o teste utilizei a palavra 'Java' de forma a ver quais skills estariam próximas da mesma. A figura 7.2 demonstra o plot realizado e por ele podemos observar que diversas tecnologias como Spring, j2ee, servlets, maven, etc., aparecem com uma similaridade muito próxima de Java. Pois tem uma relação bem forte, como por exemplo, grande parte das skills que aparecem são frameworks, linguagens, gerenciadores de pacote, dentre outras skills que são de fato utilizados apenas pela linguagem Java ou ao menos tem uma forte ligação. Assim podemos demonstrar que para a palavra Java o word embedding consegue captar a semântica das skills que são importantes para representá-la.

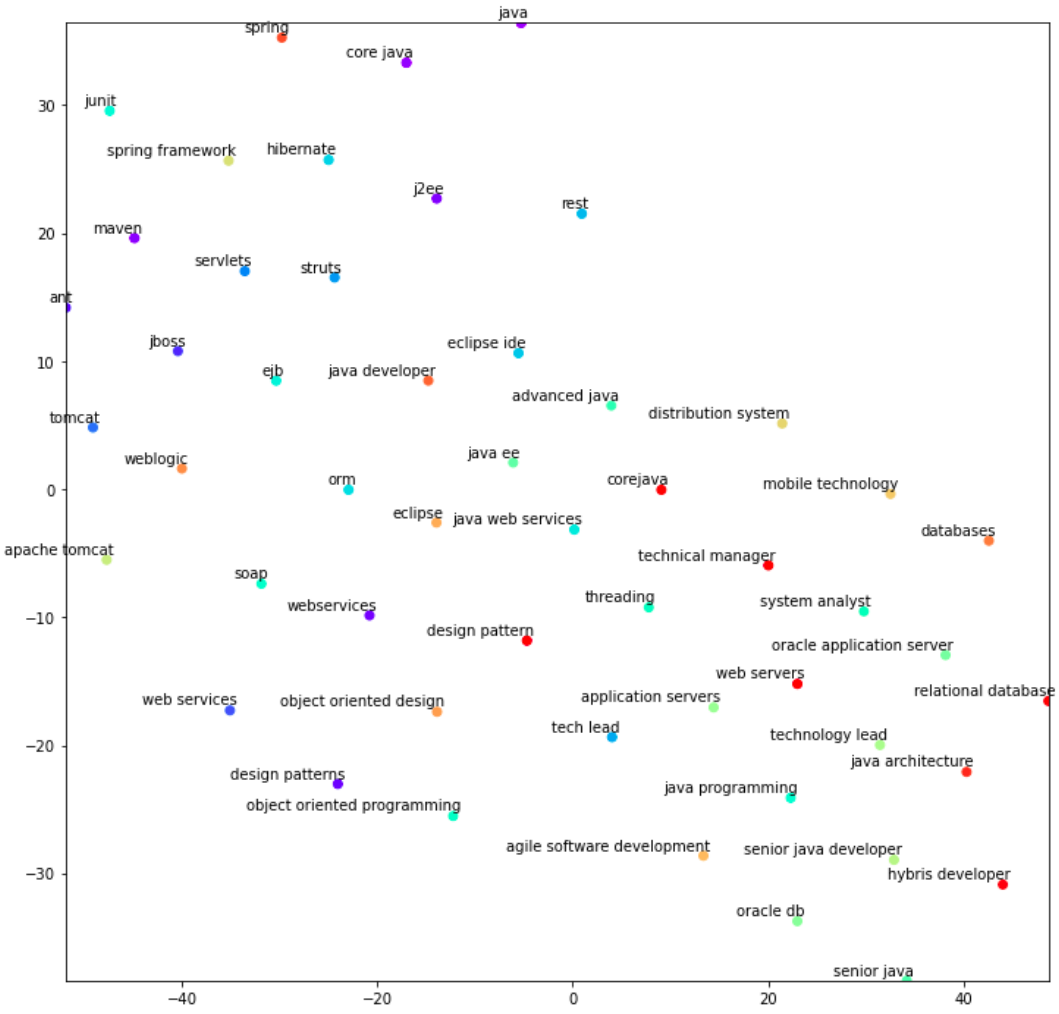

Figura 7.2: Grafico word embedding em 2D

O paper (3) incluiu alguns testes que demonstram a eficácia do modelo criado. A tabela abaixo com os resultados funciona da seguinte maneira: a função recebe uma skill como input e partir deste devolve as cinco skills cuja similaridade dos vetores é a maior. Para este projeto foi criado um modelo próprio de Word2Vec utilizando-se apenas uma parte dos dados, já que um csv mais completo não foi disponibilizado pelo Github do paper. O objetivo aqui é fazer um comparativo entre os resultados dos dois modelos, levando em conta a diferença de dados entre ambos e além disso, a existência de certos valores randômicos e possíveis hyperparameters que podem ser diferentes durante a criação do modelo. Como os resultados sempre sofrem variações toda vez que um novo word embedding é gerado, o objetivo então é observar a consistência dos resultados obtidos entre ambos. O modelo do *Aplicai* que é o modelo criado para este projeto e o Skill2Vec que é o modelo do paper (3), tem seus dados comparados na tabela abaixo:

Tabela 7.1: Resultados entre os modelos

Input	Aplicai	Skill2Vec
html5	bootstrap	css3
	css3	bootstrap
	angular	front_end
	sass	angular
	css	responsive
oop	datastructures	ood
	java-script	objective
	object_oriented	java
	distributed_system	multithread
	object-oriented_design	software debug
hadoop	hive	pig
	spark	hive
	pig	hbase
	hbase	big data
	r studio	spark
scala	eclipse modeling	zookeeper
	kafka	Spark
	nosql database	zata system
	r studio	sqoop
	impala	solrcloud
hive	hdfs	Pig
	hbase	hdfs
	flume database	hadoop
	pig	Spark
	spark	Impala

Podemos observar que na maioria dos inputs os resultados obtidos tiveram pelo menos uma ou mais skills idênticas. Em alguns casos pode existir uma variação maior entre os modelos, principalmente quando o input passado é algo genérico que pode englobar outras diversas skills. Vamos pegar por exemplo o caso da skill oop. Neste caso oop (object-oriented programming) está relacionado a um enorme número de linguagens e outras tecnologias, o que dificulta encontrar skills com um maior grau de consistência.

Vale demonstrar que mesmo assim pode-se validar que existe uma relação grande sobre os resultados pelos dois modelos. Pegando um exemplo da skill scala, pode-se observar que todas as skills são diferentes. A tabela abaixo demonstra o grau de similaridade indo de 0 até 1 (quanto mais próximo de 1

maior a similaridade) entre os resultados obtidos do input scala entre ambos os modelos:

Tabela 7.2: Resultados da comparação da similaridade das skills entre os modelos

Aplicai	Skill2Vec	similaridade
kafka	zookeeper	0.94182056
	spark	0.9086325
	sqoop	0.92537886
eclipse modeling	zookeeper	0.9117528
	spark	0.9333365
	sqoop	0.953771
impala	zookeeper	0.9185026
	spark	0.9143251
	sqoop	0.91515905

Mais testes também foram realizados com um maior número de comparações das skills. Esse resultado pode ser obtido pelo Github acessando `https://github.com/hugomachado93/aplicai-api/blob/master/notebooks/testes.ipynb`.

Foi também importante validar que o método do WMD também conseguiria bons resultados. Para isso foi utilizado os 50 mil dados do Github do Skill2Vec para realizar os testes. Com esses dados foi possível simular o cadastro de demandas com as suas skills, que ocorreriam dentro da aplicação. Logo, para um dos testes, simulando o perfil de um aluno, executamos a função `WmdSimilarity()` para ciar o modelo WMD e depois colocamos a seguinte lista de skills para realizar a busca de documentos: javascript, css e php. Após o processamento do WMD, o resultado foi como o esperado, vieram muitas skills mais ligadas ao frontend, o que demonstra a ligação com as skills que foram passadas. Abaixo coloco o resultado de dez demandas ranqueadas de acordo com a menor distância:

Tabela 7.3: Resultado das demandas ranqueadas

Demandas ranqueadas	Skills	Similaridade do documento
Demanda 1	javascript jquery php	0.802795495432378
Demanda 2	angular js php javascript	0.7913696587870684
Demanda 3	html css javascript	0.7716193904852938
Demanda 4	html css javascript	0.7645328233658393
Demanda 5	javascript css html	0.7645328233658393
Demanda 6	javascript css html	0.7645328233658393
Demanda 7	css javascript	0.7645328233658393
Demanda 8	html_and_css_and_javascript react.js javascript php mysql css	0.7417089187286581
Demanda 9	html content creating new web pages designed template html css php javascript html5 web developer	0.7324282749970482
Demanda 10	javascript css php web development web technologies	0.7320384907809082

O tempo do processamento do WMD é algo que deve ser levado em conta, pois apresenta a complexidade de $O(p^3 logp)$. Sendo p o número de palavras e considerando que palavras aqui no nosso caso são as skills, se tiver uma quantidade relativamente grande de skills isso afeta diretamente o tempo de execução do WMD. Para isso calculei o tempo médio que leva para realizar este processamento. Com diversas execuções a média foi de 35 segundos levando em conta que foram 50 mil demandas, o que já é uma quantidade relativamente grande. Esse é um fator decisivo para se ter adotado a recomendação de forma assíncrona para o usuário da aplicação, pois como mais e mais demandas vão ser cadastradas na aplicação, o tempo de execução do WMD também aumentará.

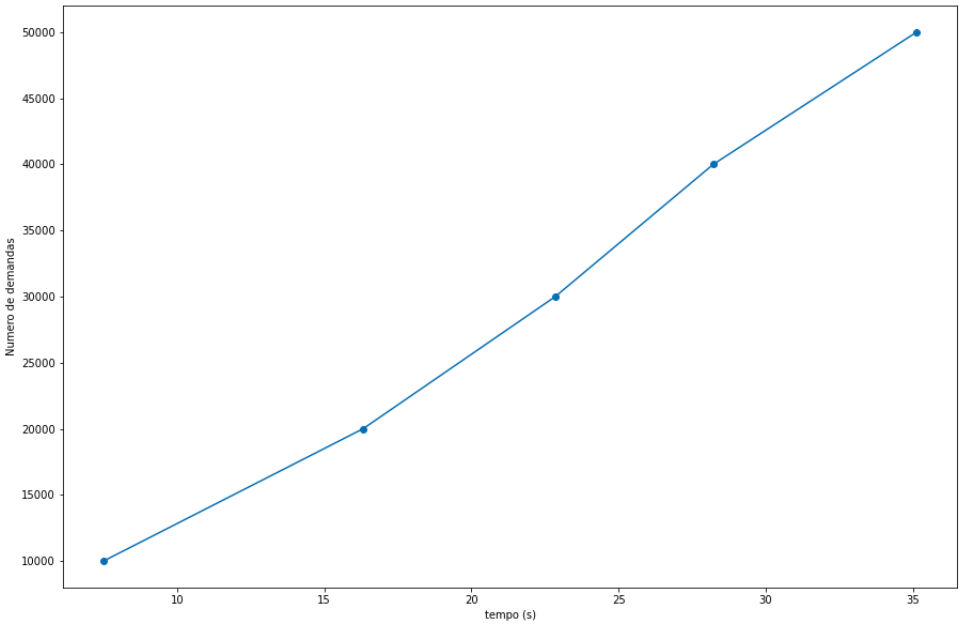

Figura 7.3: Tempo de execução do WMD de acordo com o número de demandas

8
Conclusão

O objetivo deste trabalho foi construir uma aplicação que fosse de utilidade para alunos e empreendedores. Empreendedores cadastram as demandas que precisam de ajuda para alavancar os seus negócios, ao mesmo tempo que alunos participam de demandas reais, dando a oportunidade de colocar seus conhecimentos em prática.

A construção do aplicativo com o Flutter demonstrou ser bastante eficiente. Foi possível desenvolver boas interfaces utilizando os seus componentes. O padrão BLOC que atualmente é o mais utilizado pelos desenvolvedores, possibilitou um desenvolvimento mais organizado, facilitando futuras manutenções e até mesmo a adição de novas funcionalidades. O Flutter também possibilitou gerar uma versão estável para mobile e web utilizando o mesmo código. Isto ajudou a reduzir muito o tempo de desenvolvimento para múltiplas plataformas.

A utilização do Firebase para o aplicativo colaborou para que o desenvolvimento da aplicação não demandasse tanto de um backend. Os serviços utilizados foram bastante eficientes. Com ele foi possível criar autenticação de usuários, salvar dados em bancos, invocar funções serverless e armazenar um grande volume de dados.

O diferencial da aplicação se baseou no sistema de recomendação, que utilizou o Word2Vec e o WMD. Estes possibilitaram gerar, respectivamente, um word embedding e calcular a distância de documentos. Além disso, com a ideia e os dados do paper Skill2vec foi possível validar que ambos os métodos conseguiram obter bons resultados, garantindo assim seu uso para criar o sistema de recomendação.

Grande parte do esforço deste trabalho foi achar uma boa forma de conseguir ranquear as demandas. Tanto o Word2Vec como o WMD demonstraram serem bastante eficientes para a proposta desse trabalho. Futuramente, técnicas mais modernas poderão ser utilizadas para criar um word embedding, como por exemplo o BERT (14), que é o estado da arte no quesito do NLP. Estas novas técnicas poderão ser estudadas e, se validadas adaptadas para a proposta deste trabalho.

O aplicativo ainda se apresenta em um estado de protótipo. Ele dispõe de

funcionalidades essenciais para que seja possível realizar uma prova de conceito.

Para finalizar precisamos ressaltar que o serviço do *Aplicai* tem possibilidades de melhorias, que devem ser interessantes para alunos da PUC e de outras Universidades. Uma das melhorias seria possibilitar aos usuários da aplicação a conseguirem modificar seus dados de perfil ou até mesmo da demanda. Outra possibilidade de melhoria seria quanto ao tempo de processamento que leva para executar o método do WMD. Nos testes demonstramos que com 50 mil demandas gasta-se em média 35 segundos para executar e que é possível aumentar com a chegada de mais demandas. Para melhorar este tempo de execução podemos utilizar o Relaxed Word Mover Distance demonstrado no paper(7), que pode reduzir a complexidade do WMD sem perder muito a sua eficiência. O Gensim ainda não tem disponível essa versão do WMD então, seria necessário buscar ferramentas alternativas para realizar essa melhoria.

Com todas as possibilidades de melhorias que foram discutidas acima, o aplicativo já tem uma ótima funcionalidade para as demandas dos respectivos empreendedores e alunos.

Referências bibliográficas

[1] WOEBCKEN, C.. **Push e pull marketing: o que são essas estratégias e como elas funcionam**, 2021. [Online; accessado 18-fevereiro-2021].

[2] VALVERDE-REBAZA, J. C.; PUMA, R.; BUSTIOS, P. ; SILVA, N. C.. **Job recommendation based on job seeker skills: An empirical study.** In: TEXT2STORY@ ECIR, p. 47–51, 2018.

[3] VAN-DUYET, L.; QUAN, V. M. ; AN, D. Q.. **Skill2vec: Machine learning approach for determining the relevant skills from job description**. arXiv preprint arXiv:1707.09751, 2017.

[4] WIKIPÉDIA. **Serverless computing**, 2021. [Online; accessado 25-abril-2021].

[5] WIKIPEDIA. **Mobile backend as a service**, 2021. [Online; accessado 25-abril-2021].

[6] MIKOLOV, T.; CHEN, K.; CORRADO, G. ; DEAN, J.. **Efficient estimation of word representations in vector space**. arXiv preprint arXiv:1301.3781, 2013.

[7] KUSNER, M.; SUN, Y.; KOLKIN, N. ; WEINBERGER, K.. **From word embeddings to document distances**. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, p. 957–966. PMLR, 2015.

[8] HUGHES, S.. **How we data-mine related tech skills.** URL: http://insights. dice. com/2015/03/16/how-we-data-mine-relatedtech-skills/(visited on 09/12/2017), 2015.

[9] DOMENICONI, G.; MORO, G.; PAGLIARANI, A.; PASINI, K. ; PASOLINI, R.. **Job recommendation from semantic similarity of linkedin users' skills**. In: INTERNATIONAL CONFERENCE ON PATTERN RECOGNITION APPLICATIONS AND METHODS, volumen 2, p. 270–277. SCITEPRESS, 2016.

[10] ALTSZYLER, E.; SIGMAN, M.; RIBEIRO, S. ; SLEZAK, D. F.. **Comparative study of lsa vs word2vec embeddings in small corpora: a case study in dreams database**. arXiv preprint arXiv:1610.01520, 2016.

[11] ELIA, F.. **How to compute the similarity between two text documents?**, 2020. [Online; accessado 15-março-2021].

[12] WIKIPÉDIA. **Hyperparameter (machine learning)**, 2021. [Online; accessado 20-abril-2021].

[13] WIKIPÉDIA. **Tf-idf**, 2021. [Online; accessado 01-junho-2021].

[14] HOREV, R.. **Bert explained: State of the art language model for nlp**, 2018. [Online; accessado 01-junho-2021].