PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
NÚCLEO DE EDUCAÇÃO A DISTÂNCIA
Pós-graduação *Lato Sensu* em Ciência de Dados e Big Data

Fábio Alexandre Magalhães de Holanda e Silva

Catálogo de resolução de bug de software - uma tarefa de clusterização

Belo Horizonte
2022

Fábio Alexandre Magalhães de Holanda e Silva

Catálogo de resolução de bug de software - uma tarefa de clusterização

Trabalho de Conclusão de Curso apresentado ao Curso de Especialização em Ciência de Dados e Big Data como requisito parcial à obtenção do título de especialista.

Belo Horizonte
2022

SUMÁRIO

1. Introdução

Em uma fábrica de software o processo de desenvolvimento e testes é algo constante, onde um produto é gerado por um time e depois testado por outro. Se tratando de times diferentes, podem existir muitos problemas de comunicação e feedback, principalmente quando esta fábrica de software trabalha com múltiplos projetos simultâneos onde seus contextos podem ou não ser compartilhados. Este tipo de problema ainda se agrava pelos prazos de entrega apertados de ambos os times, onde os desenvolvedores são cobrados a finalizarem suas atividades para que os testadores possam dar início a seu processo e caso algum problema seja encontrado, o desenvolvedor receberá algum feedback, ou aviso, ou até mesmo uma atividade formal para a correção daquele problema.

1.1. Contextualização

Este contexto é algo recorrente na maior parte das empresas que tem como seu produto final um software e que preza pela qualidade do mesmo, fazendo validações incrementais para que o produto chegue com o mínimo de falhas possível na mão do usuário. Porém esta atividade de desenvolver as funcionalidades, e corrigir os eventuais problemas pode ser algo extremamente repetitivo e custoso, já que vai precisar da análise do desenvolvedor para entender o problema, buscar por problemas parecidos para saber se isso já foi corrigido em algum outro projeto, aplicar a correção e validar a alteração feita. E isso vai acabar impactando no cumprimento dos prazos, fazendo com que uma entrega planejada precise ser adiada.

1.2. O problema proposto

Dado o contexto de desenvolvimento de software, correção de *bugs*, qualidade e cumprimento de prazos é perceptível que este processo de correção de *bugs* se torna um gargalo na entrega, fazendo com que o planejamento feito seja quebrado pelo tempo gasto na análise humana.

Este problema foi identificado numa fábrica de software, onde seu maior produto é a entrega do Sistema Operacional que vai embarcado nos smartphones de uma grande fabricante de eletrônicos e o tempo que é gasto na etapa de correção é crítico, já que geralmente acontece próximo do fechamento do prazo. Os dados coletados para fazer a

análise foram retirados da própria empresa em questão nos últimos 4 anos, que cedeu essas informações para a construção de uma solução viável.

1.3. Objetivos

Através desta problemática, este trabalho tem como objetivo construir uma solução que consiga categorizar e catalogar o problema entregue pelo time de teste e agrupar por grau de similaridade os problemas já revisados anteriormente, para que seja entregue informações claras de como corrigir o software para o desenvolvedor que irá corrigir o problema. Esta solução será feita baseado na análise de dados passados, retirados da própria base de atividades da empresa, a fim de treinar um modelo para que ele consiga automatizar o trabalho de análise do desenvolvedor, retornando a possível causa do erro e a provável solução.

Além do treinamento e construção de um modelo de aprendizado que consiga agrupar esses documentos, esta solução será implantada e integrada na principal ferramenta da empresa para que os times de desenvolvimento possam utilizar no seu processo tradicional de correção de *bugs.*

2. Coleta de Dados

A extração dos dados, necessários para a análise e construção da solução, foram feitas através de duas ferramentas internas da empresa para gerenciamento de atividades. Os dados estão no formato *JSON* e estão divididos em 2 *datasets*, distribuídos da seguinte maneira:

- O primeiro contendo todas as informações dos *bugs* encontrados, bem como as informações da resolução que foram feitas para corrigir o problema; e
- O segundo *dataset* contém informações sobre os projetos em questão, como a versão do sistema operacional, suportes de hardware do projeto e etc.

Esses dois *datasets* se relacionam através do nome do modelo (smartphone) em questão e a sua versão de sistema operacional. Onde cada projeto é identificado como um determinado smartphone para uma determinada versão do sistema operacional.

Foram extraídos dados dos anos de 2017 a 2021, contendo 491 registros para o *dataset* de projetos, e 8290 registros para o *dataset* de *bugs*.

Para realizar a coleta desses dados foi necessário a criação de dois scripts para fazer a consulta e a aquisição das informações. Para a construção deste script foi utilizado Python, a biblioteca *BeautifulSoup* e os módulos *requests* e *json* do Python. O portal onde detém as informações de projetos possui API própria e com isso o desenvolvimento do script foi mais simplificado, sendo necessário apenas realizar chamadas *HTTP* para a rota em questão e extrair as informações para um arquivo JSON, porém não existia documentação sobre a *API.*

Para mapear e identificar as requisições necessárias foi necessário o uso de uma ferramenta chamada *Fiddler*, que mapeia todos os pacotes realizados pelo computador. Então foi simulado o acesso ao site que vai ter suas informações extraídas para que o *Fiddler* possa capturar os pacotes que o *browser* gera. Através disto foi possível identificar o cabeçalho e corpo do pacote, e assim foi reproduzido a chamada da API que o próprio *frontend* da aplicação realizava.

```python
import json
import requests

DATA_INICIO = "2017-01-01"
DATA_FIM = "2022-01-01"
header = {
    'Accept': 'text/html, application/xhtml+xml, */*',
    'User-agent': 'Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Trident/6.0)',
    'Content-Type': 'application/x-www-form-urlencoded'
}
API = f"http://<SISTEMA_INTERNO>/projects?sd={DATA_INICIO}&ed={DATA_FIM}"

resposta = requests.get(API).json()

projetos = resposta["projects"]

open("/home/fabio.silva/dados/projetos.json", "w").write(json.dumps(projetos))
```

Figura 1: Script para coleta do *dataset* de projetos.

Já para coletar os dados dos *bugs* foi necessário aplicar a técnica de *web scrapping* para ler o *HTML* da página e capturar os dados necessários, para isso foi usado a biblioteca *BeautifulSoup* para fazer a conversão do *HTML* em um objeto estruturado, onde é possível utilizar chamadas da biblioteca para fazer consultas. Para mapear as requisições e simular os mesmos acessos do *browser,* foi utilizado novamente o *Fiddler.*

Após realizar a requisição para receber o *HTML* da página, foi necessário usar o *BeautifulSoup* para ler a tabela com os dados em questão, e extrair as informações utilizando os métodos da biblioteca (*find e find_all)*. Após, iterar sobre todas as linhas da tabela, foi gerado um arquivo *JSON* com essas informações.

```python
import requests
from bs4 import BeautifulSoup

DATA_INICIO = "2017/01/01"
DATA_FIM = "2022/01/01"
URL = f"http://<SISTEMA_INTERNO>/issueGrid/listPage?start={DATA_INICIO}&end={DATA_FIM}"

resposta = requests.get(URL)
resposta_parser = BeautifulSoup(resposta.text, "html.parser")
tabela_parser = resposta_parser.find("table", {"id": "cphMasterMain_cphMain_lblSingleID"})
```

Figura 2: Script para coleta do dataset de problemas, neste trecho está sendo feito o request via *HTTP* e convertendo o *HTML* e um objeto estruturado.

```python
problemas = []
for tr in tabela_parser.find_all("tr", {"class": "spec_issue"}):
    problemas.append({
        "title": tr.find("span", {"class": "title_issue"}).text,
        "device": tr.find("span", {"class": "device_issue"}).text,
        "os_version": tr.find("span", {"class": "os_version_issue"}).text,
        "problem": tr.find("span", {"class": "problem_description_issue"}).text,
        "category": tr.find("span", {"class": "category_issue"}).text,
        "register_date": tr.find("span", {"class": "register_date_issue"}).text,
        "resolved_date": tr.find("span", {"class": "resolved_date_issue"}).text,
        "resolved_by": tr.find("span", {"class": "resolved_by_issue"}).text,
        "cause": tr.find("span", {"class": "cause_issue"}).text,
        "resolution": tr.find("span", {"class": "resolution_issue"}).text,
    })

open("/home/fabio.silva/dados/problemas.json", "w").write(json.dumps(problemas))
```

Figura 3: Continuação do script, neste trecho está se lendo cada elemento da tabela onde contém as informações dos problemas e adicionando a uma estrutura de dados que será convertida para JSON.

Ao final do processo de extração dos dados, obteve-se um *dataset* de projetos contendo 491 registros e um *dataset* de *bugs* contendo 8290 registros. As informações de cada campo do *dataset* está descrito nas tabelas abaixo:

Nome do atributo	**Descrição do atributo**	**Tipo de dado**
title	Título do problema	Texto
device	Nome do dispositivo	Texto
os_version	Versão do sistema operacional	Texto

problem	Descrição do problema apresentado pelo time de teste	Texto
category	Categoria do problema definida pelo time de desenvolvimento após a finalização do problema	Texto
register_date	Data de registro do problema	Data
resolved_date	Data de correção do problema	Data
resolved_by	Desenvolvedor que corrigiu o problema	Texto
resolution	Descrição da resolução do problema, feita pelo desenvolvedor	Texto
cause	Descrição da causa do problema, feita pelo desenvolvedor	Texto

Tabela 1: Descrição do *dataset* de problema.

Nome do atributo	**Descrição do atributo**	**Tipo de dado**
device	Nome do dispositivo	Texto
os_version	Versão do sistema operacional	Texto
network_support	Qual tipo de rede este dispositivo suporta	Texto (5G/4G/3G/WIFI)
memory	Quantidade de memória RAM este dipositivo possui	Número
storage	Quantidade de armazenamento interno este dispositivo possui	Número
chipset	Tipo de chipset que este dispositivo possui	Texto (Qualcomn, Mediatek, LSI)
release_date	Data de lançamento do projeto para o mercado	Date

Tabela 2: Descrição do *dataset* de projeto.

3. Processamento/Tratamento de Dados

Foram realizados uma série de alterações e tratamento nos *datasets* coletados para que sua análise e utilidade se torne mais viável nas próximas etapas.

Ambos os *datasets* estão em formato *JSON* e para o tratamento devido foi utilizado a biblioteca *pandas*, para uma análise prévia e manipulação dos dados. Em primeiro momento foi feita uma busca por dados omissos nas coleções e foi identificado que o único parâmetro que apresenta dados omissos na coleção de projetos é o *storage*. Para preencher essa informação, foi calculado a média de todos os registros de *storage*, já que é um campo numérico, e com isso é realizada a alteração.

```
import pandas as pd

df_projetos = pd.read_json("/home/fabio.silva/dados/projetos.json")
storage_media = df_projetos[["storage"]].means(axis=0)
df_projetos.storage.fillna(value=storage_media, inplace=True)
```

Figura 4: Script para substituir os dados faltantes em *storage* com a média das informações que existem no *dataset.*

Já no *dataset* de *bugs* foi identificado que alguns registros não tinham informação de causa nem de resolução, e como estes campos são textuais, contendo descrições do problema, então os registros iriam ficar inutilizados, e para não atrapalhar no processo de aprendizado, foi removido esses registros.

```
import pandas as pd

df_problemas = pd.read_json("/home/fabio.silva/dados/problemas.json")

df_problemas.dropna()
```

Figura 5: Script para remoção dos registros com colunas vazias.

Após isto, foi analisada a variabilidade dos textos no campo *category*, no *dataset* de *bugs*, e foi identificado que o mesmo termo aparece de diferentes formas, variando pontuação, acentuação, abreviação e etc. Para isso foi criado um dicionário com os prováveis

termos e sua representação formalizada, para que diminua a quantidade de termos similares.

```
TERMOS_SIMILARES = {} # mascarado para evitar vazamento de dados

df_problemas = pd.read_json("/home/fabio.silva/dados/problemas.json")

for termo in TERMOS_SIMILARES:
    df_problemas.loc[df_problemas['category'] in TERMOS_SIMILARES[termo]].replace(value=termo, inplace=True)
```

Figura 6: Script para padronização da *category* através do dicionário de termos (o dicionário foi mascarado para evitar vazamento de informação).

O campo *resolved_by* no *dataset bugs* contêm informações dos funcionários, por questão de anonimato esta coluna foi removida, já que também não ajudaria nas análises em questão.

```
df_problemas.drop('resolved_by', axis=1, inplace=True)
```

Figura 7: Remoção da coluna *resolved_by*.

E por fim, os campos de data (*resolved_date*, *register_date* e *release_date*) estavam em texto plano, e para realizar análise e filtragem eles precisam estar no formato de data, para isso foi feito uma conversão para ambos os *datasets*.

```
df_problemas["resolved_date"] = pd.to_datetime(df_problemas["resolved_date"], format="%Y-%m-%d")

df_problemas["register_date"] = pd.to_datetime(df_problemas["resolved_date"], format="%Y-%m-%d")

df_projetos["release_date"] = pd.to_datetime(df_projetos["release_date"], format="%Y/%m/%d")
```

Figura 8: Script para converter as datas em texto plano para um formato *datetime.*

Ao concluir o tratamento dos dados das duas coleções, precisamos unir as informações em um único arquivo, para que as próximas etapas possam realizar seus processos de uma forma mais simplificada. A coluna em comum entre os *datasets* são *os_version* e *device*, atributos estes que são a chave composta para projetos. Ao unir as duas

entidades obtivemos como resposta um *dataset* com 8290 registros. Para finalizar foi exportado o objeto tratado para um arquivo *JSON* onde será utilizado nas etapas seguintes.

```
df_problemas_projetos = pd.merge(df_problemas, df_projetos, on=["device", "os_version"])

data = df_problemas_projetos.to_json(orient='table')

open("/home/fabio.silva/dados/problemas_projetos.json", "w").write(json.dumps(data))
```

Figura 9: Script para unir os dois *DataFrames* através das colunas de *device* e *os_version*. Após a união, o objeto será exportado para um arquivo *JSON*.

4. Análise e Exploração dos Dados

As análises feitas nessa etapa foram realizadas utilizando, sobretudo, a experiência e o conhecimento prévio dos dados abordados e observando padrões textuais nas descrições dos problemas reportados pelo time de teste.

Existem obviamente, diferenças na escrita de um testador para outro, porém aparentemente existe um padrão/template que é seguido, onde podemos perceber que problemas reportados em um mesmo módulo ou funcionalidade, tem um padrão de escrita muito específica. Por exemplo, problemas reportados no módulo de exibição da iconográfica da rede de dados conectada, como por exemplo, está sendo exibido o ícone "4G" no smartphone, e o correto seria "LTE", tem uma descrição com palavras chaves fundamentais, que servem como identificador e categorizador do problema.

"*DUT have not been set in config D, for the carrier X and Y*
According to the requirement of the carriers X and Y, the icon 5G should have the option D
-DUT under Coverage 5G X and Y,
-5G icon only appears when connected to the NR and ULI=1.
-if not connected to NR shows 4G icon.
DUT must be customized with Config D"

Neste texto, percebemos algumas palavras importantes para definição do problema, como por exemplo: *icon*, 5G, 4G, NR, config D, UL1=1. E percebemos também palavras que não vão ajudar no trabalho de treinamento e aprendizado do modelo, como conectivos, preposições, artigos ou palavras comuns como *customized*. Então essas palavras deverão ser removidas da base para que o treinamento possa ocorrer de forma mais performática.

Para isto, precisamos criar um *Bag Of Words* (repositório de termos relevantes para o contexto) com palavras chaves relevantes que serão reforçadas no texto para o contexto analisado e remover tudo aquilo que fizer parte do nosso repositório de palavras chaves indesejadas, o nosso *Stop Words*. Esses repositórios foram gerados baseado na experiência do processo e com a ajuda dos times de desenvolvimento, que incrementaram com palavras relevantes para os seus contextos, lembrando que cada time de desenvolvimento possui seu próprio conjunto de termos e palavras chaves.

Com isso, precisamos incrementar na fase de tratamento dos dados adicionando esta alteração no *dataset*.

```
import nltk
STOP_WORDS_ENGLISH = nltk.corpus.stopwords.words('english')
STOP_WORDS_REQUIREMENT = ['behavior', 'reproduce', 'up', 'set', 'on', 'dut', 'starting', 'expected', 'first', 'call',
                          'same', 'as', 'carrier', 'you', 'it', 'that', 'applied', 'to', 'for', 'change', 'when',
                          'requirement', 'second', 'steps', 'be', 'must', 'is', 'according', '3.', 'after',
                          'requirements',
                          'files', 'information', 'phone', 'this', 'new', 'should', 'with', 'list', 'before', 'have',
                          'pre', 'and', 'does', '2.', 'was', 'cr', 'sample', '1.', 'are', 'display', 'check', 'select',
                          'only', 'wrong', 'sw', 'condition', 'attached', 'using', 'current', 'insert', 'the', 'start',
                          'device', 'but', 'of', 'from', 'document', 'in', 'not', 'a', 'after', 'selecting', 'whole',
                          'process', 'again', 'makes', 'hot', 'it', 'route', 'problem', 'expected', 'result',
                          'reproduction', 'does', 'not', 'show', 'when', 'using', 'sim', 'card', 'cards', 'code',
                          'version', 'build', 'id', 'sku', 'test', 'flash', 'binaries', 'category', 'setup', 'all',
                          'item', "o", "de", "do", "e", "os", "com",
                          'request', 'cited', 'complete', 'screen', '0', '1', '2', '3', 'dut.', '&', 'reproduce1',
                          'bellow2', 'u\\sel', '|'] + STOP_WORDS_ENGLISH
```

Figura 10: Construção das listas de *Stop Of Words*.

```
BAG_OF_WORDS = [] # mascarado para proteção a dados internos

problemas = json.loads(open("/home/fabio.silva/dados/problemas.json", "r").read())

for problema in problemas:
    problema["problem_train"] = " ".join([word for word in problema["problem"].lower().split() if word not in STOP_WORDS_REQUIREMENT])
    problema["resolution_train"] = " ".join([word for word in problema["resolution"].lower().split() if word not in STOP_WORDS_REQUIREMENT])
    problema["cause_train"] = " ".join([word for word in problema["cause"].lower().split() if word not in STOP_WORDS_REQUIREMENT])

for problema in problemas:
    problema["problem_train"] = " ".join([f"{word} {word} {word}" for word in problema["problem_train"].lower().split() if word in BAG_OF_WORDS])
    problema["resolution_train"] = " ".join([f"{word} {word} {word}" for word in problema["resolution_train"].lower().split() if word in BAG_OF_WORDS])
    problema["cause_train"] = " ".join([f"{word} {word} {word}" for word in problema["cause_train"].lower().split() if word in BAG_OF_WORDS])
```

Figura 11: Remoção das palavras contidas no *Stop Of Words* e reforço das palavras no *Bag Of Words*.

Com isto, é possível analisar e concluir que a utilização de um modelo de agrupamento seria interessante para o caso de uso em questão, onde existem características semelhantes entre os dados do conjunto e com isso juntá-los por essas similaridades ajudará o desenvolvedor a analisar problemas similares ao seu, para a partir disto possuir várias resoluções de problemas que podem ser aplicados no seu caso em questão.

Com este modelo treinado, esperamos que o tempo de correção de um problema seja reduzido, visto que possivelmente uma solução já será dada como resposta ao agrupamento que o modelo irá gerar.

5. Criação de Modelos de Machine Learning

Os experimentos feitos em aprendizado não supervisionado utilizando *clusters* foram feitos em *Python*, através da ferramenta *Google Colaboratory*.

A ideia do trabalho é identificar agrupamentos de atividades, geradas pelo time de testes, pela análise textual da descrição do problema, bem como as características do projeto em questão, para que com isso tenhamos um grupo de possíveis soluções para o problema em estudo. Tal análise é dificultada pelos parâmetros do modelo serem em sua maioria por texto. Por exemplo, a descrição do problema, que é definida pelo testador, é um dos atributos do *dataset bug* e está escrito em texto plano.

Este tipo de atributo precisará ser convertido por uma representação numérica para que ajude o modelo a realizar o processamento de aprendizado. Por outro lado, existem também atributos textuais que são fáceis de categorizar, como por exemplo o atributo *network_type* que representa qual tipo de rede de dados o projeto suporta, podendo ser 5G, 4G, 3G ou Wifi.

Então, antes de aplicar os dados a um modelo de aprendizado, primeiro seria necessário criar um vetor com uma representação numérica do texto encontrado no *dataset*. Esta representação seria o *TF-IDF*, que é uma representação estatística que tem o intuito de indicar a importância de uma palavra de um documento em relação a uma coleção de documentos. E para fazer esta conversão foi utilizada a biblioteca em *Python* chamada *scikit-learn*, mais precisamente o módulo *TfidfVectorizer*.

```
from sklearn.feature_extraction.text import TfidfVectorizer

texto_problemas = []  # mascarado para evitar vazamento de informação
vector_texto_problemas = TfidfVectorizer()
vector_texto_problemas.fit([texto_problemas])
```

Figura 12: Treinando um vetor TF-IDF com todos os termos encontrados em todos os documentos do dataset.

```
dataset = json.loads(open("/home/fabio.silva/dados/problemas_projetos.json", "r").read())
resultado = []
for item in dataset:
    texto = " ".join([item[key] for key in item])
    resultado.append({
        "id": item["id"],
        "vector": vector_texto_problemas.transform([texto]).toarray()[0]
    })

open("/home/fabio.silva/dados/problemas_projetos_vetorizado.json", "w").write(json.dumps(resultado))
```

Figura 13: Abrindo o *dataset* e gerando um novo conjunto de dados formado pelo *ID* e pelo vector gerado pelo objeto *TF-IDF* vetorizado dos documentos.

Após a conversão, iremos aplicar o *dataset* ao nosso primeiro modelo de estudo, o *K-means*. Para as nossas validações, quantidade de *clusters* e hiper parâmetros usados, vamos usar o próprio time de desenvolvimento para validar as sugestões de agrupamento encontrados, para garantir uma maior confiabilidade na acurácia dos grupos.

Os modelos gerados tiveram seu número de *clusters* variando baseado na quantidade de categorias encontrada no atributo *category* do *dataset bug*. Este atributo tem 9 dados divergentes, por isso supomos que no máximo podemos encontrar 9 grupos diferentes. Porém, estes grupos podem ser tão pequenos que acabam não sendo relevantes para o modelo, e acaba sendo absorvido por um outro cluster com uma certa semelhança, ou podemos acabar encontrando agrupamentos que não tínhamos conhecimento, então vamos variar o *cluster* de 4 a 14.

```python
from sklearn.cluster import KMeans
from joblib import dump
dataset = json.loads(open("/home/fabio.silva/dados/problemas_projetos_vetorizado.json", "r").read())
X = []
for item in dataset:
    X.append(item["vector"])

for k in range(4, 15):
    kmeans = KMeans(n_clusters=k, random_state=0).fit(X)
    resultado = []
    for index, label in enumerate(kmeans.labels_):
        resultado.append({
            "id": dataset[index]["id"],
            "cluster": label
        })
    dump(kmeans, f"/home/fabio.silva/dados/kmeans_{k}.joblib")
    open(f"/home/fabio.silva/dados/resultado_kmeans_{k}.json", "w").write(json.dumps(resultado))
```

Figura 14: Treinamento do modelo *K-Means* usando o número de clusters variando de 4 a 14. O resultado gerado foi exportado para um arquivo JSON guardando os agrupamentos obtidos, e foi utilizado a biblioteca *joblib* para persistir o modelo treinado.

Para armazenar os resultados encontrados, foi gerado para cada registro o grupo correspondente e com isso foi criado um novo arquivo contendo o ID e o número do cluster. Este arquivo será usado pelo time de desenvolvimento para validar os agrupamentos variando o número de *clusters* e através desta análise, será definido a melhor quantidade de grupos.

Para a segunda análise foi utilizado o algoritmo *Birch*, também fornecido pela biblioteca *scikit-learn*. Este modelo é baseado em sub-clusters, onde ele constrói uma árvore chamada *CFT* (Clustering Feature Tree) para os dados fornecidos. Os dados são essencialmente compactados em um conjunto de nós. Os nós possuem vários sub-clusters, e esses sub-clusters podem ter outros nós como filhos. Este modelo possui dois importantes hiper parâmetros, a quantidade de cluster (*n_clusters*) e o *branching_factor*, que é o número máximo de subcluster em cada nó. Iremos fazer nossos testes mudando apenas o *n_clusters*, similar a solução usada no *K-means*.

```python
from sklearn.cluster import Birch
from joblib import dump
dataset = json.loads(open("/home/fabio.silva/dados/problemas_projetos_vetorizado.json", "r").read())
X = []
for item in dataset:
    X.append(item["vector"])

for k in range(4, 15):
    birch = Birch(n_clusters=k).fit(X)
    resultado = []
    for index, label in enumerate(birch.labels_):
        resultado.append({
            "id": dataset[index]["id"],
            "cluster": label
        })
    dump(birch, f"/home/fabio.silva/dados/birch_{k}.joblib")
    open(f"/home/fabio.silva/dados/resultado_birch_{k}.json", "w").write(json.dumps(resultado))
```

Figura 15: Treinamento para o modelo *Birch.*

Para a última análise foi utilizado o modelo *SpectralClustering*, onde sua principal característica é usar uma projeção laplaciana nos clusters. Para padronizar os testes, foi utilizado o mesmo número de clusters.

```python
from sklearn.cluster import SpectralClustering
from joblib import dump
dataset = json.loads(open("/home/fabio.silva/dados/problemas_projetos_vetorizado.json", "r").read())
X = []
for item in dataset:
    X.append(item["vector"])

for k in range(4, 15):
    spectral = SpectralClustering(n_clusters=k, assign_labels='discretize', random_state=0).fit(X)
    resultado = []
    for index, label in enumerate(spectral.labels_):
        resultado.append({
            "id": dataset[index]["id"],
            "cluster": label
        })
    dump(birch, f"/home/fabio.silva/dados/spectral_{k}.joblib")
    open(f"/home/fabio.silva/dados/resultado_spectral_{k}.json", "w").write(json.dumps(resultado))
```

Figura 16: Treinamento do modelo *SpectralClustering.*

Os testes foram realizados da mesma forma para os 3 algoritmos em análise. Para os testes foram utilizados 4 times de desenvolvimento com 25 pessoas cada. Com isso foi gerado cerca de 100 avaliações para cada iteração de cada modelo de aprendizado, resultando em cerca de 11 testes para cada modelo de aprendizado em estudo. A bateria de testes durou cerca de 2 meses para ser concluída. Para os desenvolvedores poderem visualizar os grupos formados, foi utilizado a própria biblioteca *pandas* para filtrar cada grupo e visualizar os dados em *JSON.*

6. Interpretação dos Resultados

Analisando o resultado dos testes, é possível perceber a baixa performance dos algoritmos *k-means* e *SpectralClustering* para agrupar os dados em questão. Em geral, muitos grupos ficavam com poucos dados e alguns outros grupos apresentavam muitos registros, sendo eles bem diversos.

Mesmo controlando a quantidade de grupos gerados, os resultados não foram satisfatórios. Melhoras foram refletidas apenas quando diminuía o número de clusters, porém com poucos clusters o problema em questão levantado por esse trabalho não era resolvido, já que agrupava muitos dados com características divergentes, apenas com termos similares que não indicavam um grupo. Isso acontece pela vizinhança entre os grupos, que apresentavam certa similaridade, que não era relevante para o problema abordado.

Porém usando o modelo *Birch*, podemos observar um grande salto de acurácia na geração dos clusters. Os resultados dos testes para este modelo foram melhores do que qualquer número de cluster testado. O melhor resultado encontrado foi para 9 clusters.

Clusters	**Resultado K-Means**	**Resultado SpectralClustering**	**Resultado Birch**
4	6.4	7.41	7.11
5	6.2	6.77	7.44
6	5.11	4.31	7.23
7	5.34	2.11	8.14
8	4.56	1.44	8.59
9	4.14	0	9.12
10	4.12	0	9.01
11	3.33	0	8.34
12	2.14	0	8.13
13	0	0	7.53
14	0	0	6.55

Tabela 3: Resultado para os testes dos modelos de aprendizado.

Os testes foram feitos com uma nota de 0 a 10 para cada modelo gerado. Cada desenvolvedor analisou os dados em questão para cada grupo formado em cada teste

executado e através disso ele deu uma nota. Para gerar um resultado foi feito uma média das notas. Dado os resultados encontrados foi escolhido o modelo *Birch* com 9 clusters para ser usado na solução desenvolvida para este trabalho.

7. Apresentação dos Resultados

O trabalho de desenvolvimento, teste e correção de software é um processo constante, que durante a concepção de um produto é algo cíclico, onde irá consumir muito esforço e tempo dos funcionários. Porém, em muitos desses processos, como a execução de testes, existem ferramentas para auxiliar na automação e otimização desse processo. Mas não encontramos o mesmo na etapa de correção de *bugs.* Onde muitas vezes contamos apenas com ferramentas que tentam apontar algum grau de similaridade baseado na ocorrência dos termos, o que nos dá um resultado na maioria das vezes irrelevante. Ou seja, não nos ajuda a saber qual a forma de corrigir um *bug* dado sua descrição.

É a partir deste problema que este trabalho foi desenvolvido, com o objetivo de entregar um modelo de aprendizado não supervisionado, onde todos os problemas conhecidos pela empresa, serão treinados e agrupados, para que um dado novo problema, seja adicionado a algum grupo, e através disto o desenvolvedor terá como resultado um grupo completo com *bugs* já corrigidos, com suas categorias e soluções.

Este trabalho foi aplicado a uma empresa que trabalha com desenvolvimento de Android Embarcado, implementando customizações para operadoras de telefonia móveis da américa latina, para uma grande fabricante de eletrônicos. Este trabalho consiste em extrair os dados das diversas fontes descentralizadas da empresa, minerar e limpar esses dados, estudar as informações adquiridas e a partir disso treinar modelos de aprendizado não supervisionado e comparar seus resultados utilizando o próprio time de desenvolvimento para avaliar e ranquear os modelos em questão.

Atualmente a empresa possui um sistema web interno, onde é utilizado para checar e resolver os *bugs.* Esta ferramenta possui alguns utilitários e é a principal interface gráfica que é utilizada para realizar o desenvolvimento e correção de software. E foi através disso que foi feita a inserção do modelo adquirido no capítulo anterior, para que seja adicionado a funcionalidade do desenvolvedor poder receber sugestões de correção para o problema, a partir do agrupamento gerado pelo modelo de aprendizado.

Esta ferramenta é desenvolvida em *Python* utilizando o framework web *Django.* E para fazer este acoplamento, foi necessário criar uma *API REST* em *Django,* onde irá receber

uma requisição *HTTP* com os dados do *bug* em questão. Este *bug* será tratado para fazer a conversão utilizando o algoritmo *TF-IDF* e com isso será entregue ao modelo *Birch* onde será dado qual o agrupamento ele pertence. Este grupo será retornado para a página web através de um *JSON*, que por sua vez será exibido na tela do usuário.

```
def find_similarity(request):
    from joblib import load

    texto_problemas = []  # mascarado para evitar vazamento de informação
    vector_texto_problemas = TfidfVectorizer()
    vector_texto_problemas.fit([texto_problemas])

    data_problem = request.POST.dict().get("problem", "")
    data_problem_vectorizer = list(vectorizer_problem.transform([data_problem]).toarray()[0])

    birch = load(f"birch_9.joblib")
    group_item = int(birch.predict([data_problem_vectorizer])[0])

    result = clustering.labels_[group_item]
    return JsonResponse({"result": result})
```

Figura 17: REST API para fornecer a funcionalidade de agrupar um dado *bug* passado como parâmetro. Esta *API* está integrada com a principal ferramenta da empresa.

Para fazer o consumo da *API* e exibição das informações, foi desenvolvido um módulo em *ReactJS* (biblioteca para desenvolvimento de aplicações web). As sugestões estão sendo exibidas em uma tabela, onde o desenvolvedor pode consultar informações relacionadas a causa raiz, contramedida, resolução do problema e a categoria do problema.

Através dessas informações, o funcionário pode resolver o problema, apenas se baseando no seu agrupamento.

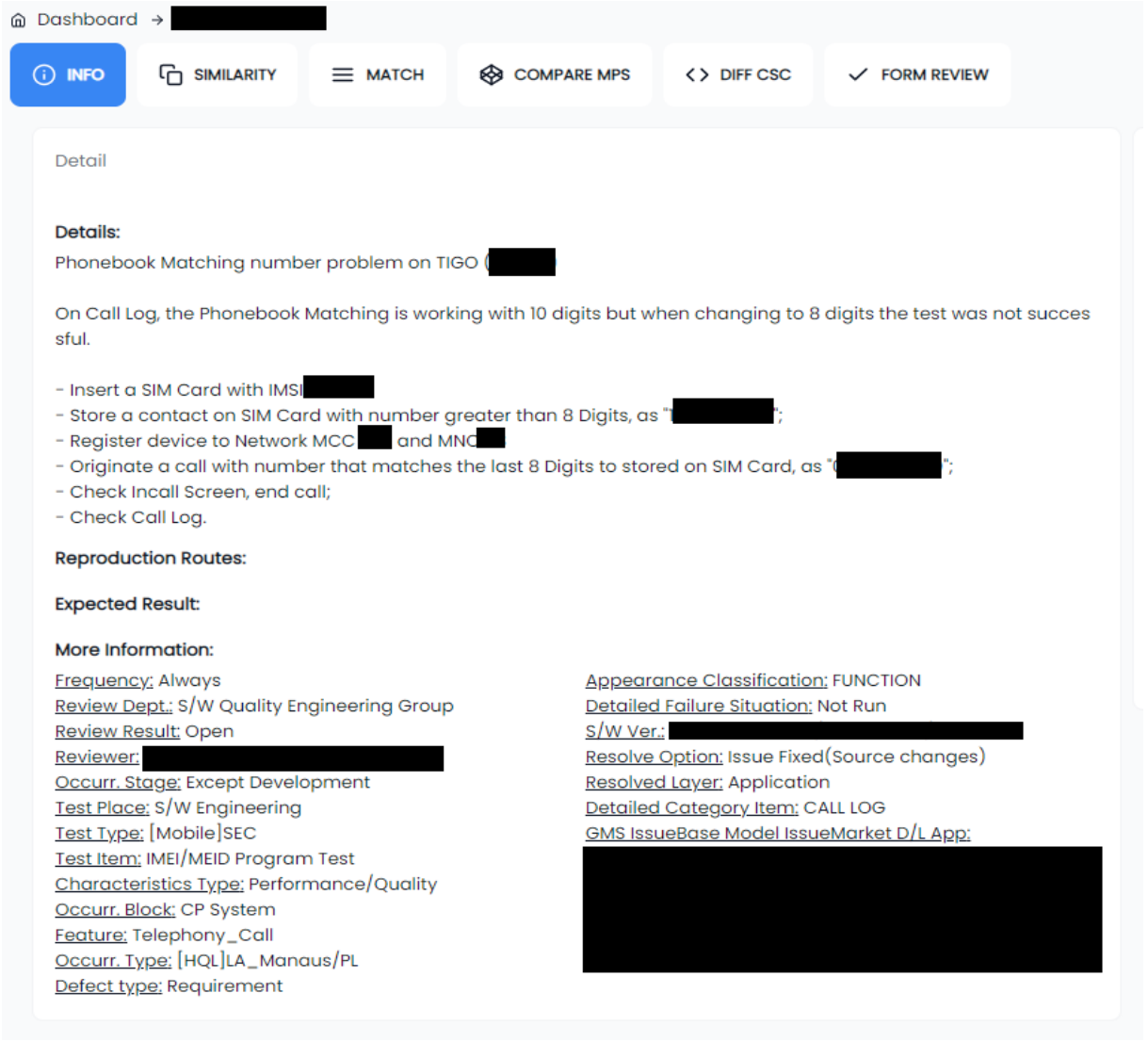

Figura 18: Tela da ferramenta na qual a solução apresentada neste trabalho foi integrada. Esta tela está mostrando a descrição de um dado *bug* reportado pelo time de teste. Algumas informações foram ocultadas para preservar informações sensíveis.

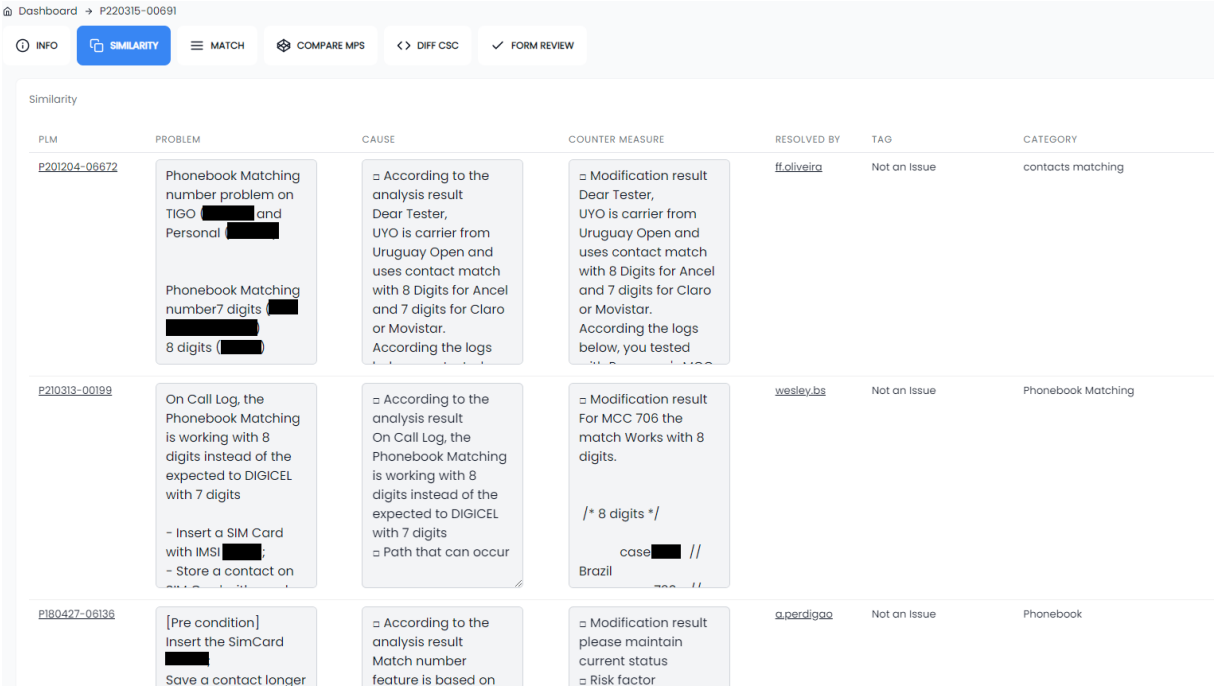

Figura 19: Agrupamento retornado pela *API* criada. Este agrupamento foi gerado pelo modelo treinado no capítulo anterior usando o algoritmo *Birch.*

Essa solução já está sendo utilizada pelo time de desenvolvimento para resolver *bugs.* Este módulo foi posto em produção em 15 de outubro de 2021, e realizando medições no tempo de execução de uma atividade para correção de problemas foi identificado uma queda média de 60%. Para identificar e resolver o *bug,* reportado pelo time de teste, se gastava em média 8 horas. Hoje esse tempo caiu para 3 horas e 30 minutos. Pelo sucesso da solução, o módulo foi adotado como processo padrão para identificação e correção de um *bug.*

Os resultados encontrados ainda podem ser melhorados ajustando os hiper parâmetros dos algoritmos e ainda refinando o nosso *Bag of Words* com palavras mais relevantes para o contexto em questão. Vale ressaltar, que o conjunto de termos relevantes varia de time para time. Onde o time de desenvolvimento que trabalha com protocolo de rede possui um repositório de palavras chaves diferentes de quem trabalha com desenvolvimento de aplicação *mobile.*

8. Links

Link para o vídeo: https://www.youtube.com/watch?v=hFzKHaPZtmw

Link para o repositório: https://github.com/fabmax94/puc-minas-data-science-tcc

Link para os *datasets:*

https://raw.githubusercontent.com/fabmax94/puc-minas-data-science-tcc/main/problemas.json

https://raw.githubusercontent.com/fabmax94/puc-minas-data-science-tcc/main/projetos.json

REFERÊNCIAS

[1] Python. Disponível em <https://python.org/>.

[2] Pandas. Disponível em <https://pandas.pydata.org/>.

[3] K-Means. Disponível em <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>.

[4] Birch. Disponível em <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.Birch.html#sklearn.cluster.Birch>.

[5] BIRD, Steven, Edward Loper e Ewan Klein, *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[6] SAWANT, Mrunal. Truncated Singular Value Decomposition (SVD) using Amazon Food Review. Disponível em <https://medium.com/swlh/truncated-singular-value-decomposition-svd-using-amazon-food-reviews-891d97af5d8d>.

[7] TfidfVectorizer. Disponível em <https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html>.

[8] Clustering. Disponível em <https://scikit-learn.org/stable/modules/clustering.html>.

[9] Model Persistence. Disponível em <https://scikit-learn.org/stable/modules/model_persistence.html>.

[10] Google Colab. Disponível em <https://colab.research.google.com/>.