Trabalho prático 1 - GRADI
Bubble gun

Mateus Pinto da Silva[1]

[1]Ciência da Computação – Universidade Federal de Viçosa (UFV-caf) – Florestal – MG – Brasil

(mateus.p.silva[1])@ufv.br

Resumo. *Redes sociais geram bolhas sociais que ameaçam a democracia cada vez mais. O Twitter é a rede social mais politizada, e consequentemente a que mais apresenta tais bolhas. O Bubble gun é um sistema de análise de sentimentos para o Twitter, com foco na análise dos tweets dos presidenciáveis das eleições de 2022. O sistema é uma API Rest que faz uso extenso de JSON, e permite o download de suas informações em três formatos semiestruturados: o já citado JSON, o XML e o CSV. Todas as informações são salvas em um banco de dados relacional. Infelizmente, devido ao curtíssimo tempo, muitas coisas não puderam ser implementadas, e elas serão explicadas no decorrer deste documento.*

1. Como executar

Para executar, é necessário ter o interpretador Python 3, o gerenciador de bibliotecas pip e o ambiente de virtualização virtualenv. Ambos podem ser baixados no site oficial da linguagem. Todo o trabalho foi testado utilizando o sistema operacional Linux Ubuntu 20.04 LTS através da máquina virtual WSL, então o sistema deve funcionar bem em qualquer Linux nativo ou virtualizado. Considerando a natureza interpretada da linguagem Python, o sistema provavelmente poderá ser executado sem problemas também no Windows e no MacOS.

Com todos os requisitos cumpridos, basta abrir esta pasta utilizando o terminar e utilizar o seguinte comando para instalar as bibliotecas necessárias:

```
virtualenv venv
source venv/bin/activate
pip install -r requirements.txt
```

Depois, no mesmo terminal, execute o próximo comando para executar o backend da sistema:

```
make
```

Em outro terminal na mesma pasta, execute o seguinte comando para executar o frontend do sistema:

```
make run
```

Figura 1. O frontend sendo acessado no browser Microsoft Edge.

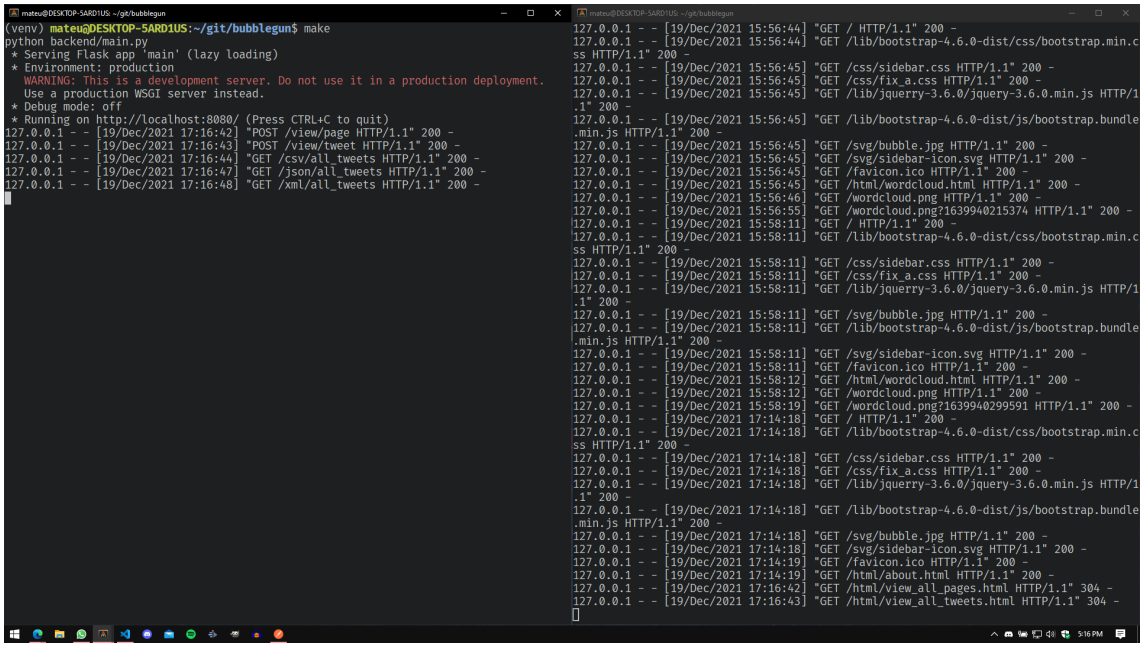


Figura 2. O backend sendo executado no terminal a direita, e o front a esquerda.

2. A ideia inicial

O sistema Bubble gun seria um analisador de bolhas, capaz de baixar dinamicamente os tweets dos presidentes e seus comentários através de uma Rest API em JSON, analisar os sentimentos imbuídos, armazenar tais dados com as análises num banco de dados relacional PostgreSQL, e servir tudo isso através da uma API Rest em Flask que permita acessar tais informações com gráficos e permitir o download das informações em três formatos semiestrutudados: CSV, JSON e XML.

Tal análise de sentimento seria feita através das bibliotecas NLTK e SKLearn do Python, e a API do Twitter seria acessada através da biblioteca Requests. O frontend seria feito em Angular.

Tecnologia	**Utilidade**
Requests + Twitter API	Baixar os tweets e comentários dos presidenciáveis automaticamente
Flask	Biblioteca para criação de API Rest em Python
NLTK, SKlearn	Bibliotecas para análise de sentimentos
PostgreSQL	Banco de dados relacional
Angular	Framework para criação de frontend.

3. O que foi implementado

Infelizmente, devido ao curtíssimo tempo, pouca coisa foi implementada. A API do Twitter não pôde ser usada devido a limitações. A análise de sentimentos também não foi implementada por falta de tempo, visto que levaria dias para classificar alguns tweets para treinar o modelo. Optei também por utilizar SQLite ao invés do PostgreSQL para evitar problemas de conexão e a possível necessidade de Docker (ou alguma outra virtualização de máquinas virtuais, redes e portas). Também optei por um frontend muito simples, utilizando Jquery ao invés de Angular.

Tecnologia	**Utilidade**
NENHUMA (MANUAL)	Tweets precisam ser colocados manualmente.
Flask	Biblioteca para criação de API Rest em Python
NENHUMA (MANUAL)	Análise de sentimentos precisa ser manual.
SQLITE	Banco de dados relacional simplificado.
JQUERY	Biblioteca de frontend para conexão com API Rest.

4. A persistência dos dados

Como o trabalho foi implementado de forma super simplificada, isso foi refletido no banco de dados, que conta com apenas duas tabelas. Uma guarda as páginas (ou seja, os usuários do Twitter) e a outra os Tweets.

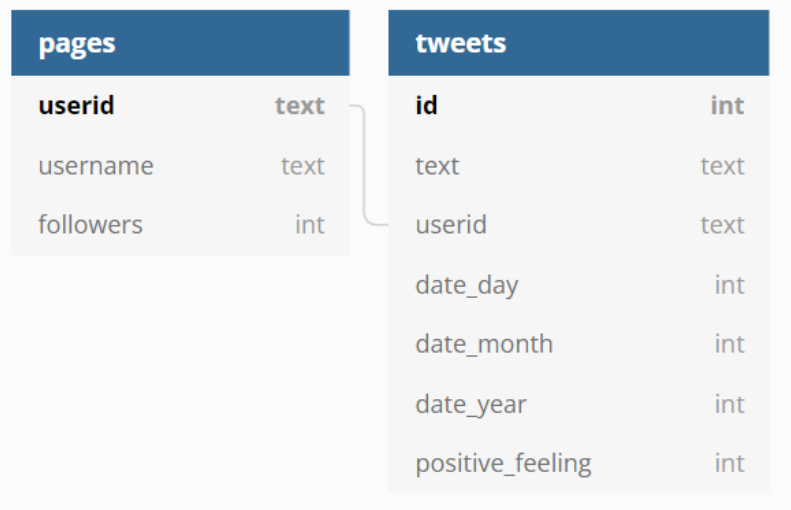


Figura 3. O diagrama do banco de dados.

5. Os dados semiestruturados

Todo o trabalho lidou com os dados semiestruturados levando em consideração a sua estrutura, ou seja, utilizando o método inlining. Isso ocorreu devido a natureza dos dados, em que sempre era possível saber o esquema desses dados à priori. Devido a API Rest, todos os retornos das rotas são JSON, como por exemplo:

```
@APP.route('/view/tweet', methods=['POST'])
def view_tweet():
    DB_CUR = get_database().cursor()
    return {'error': 0, 'response': {'tweets': [{'userid': x[0], 'text'
    :x[1], 'date_day':x[2], 'date_month':x[3], 'date_year':x[4], '
    positive_feeling':x[5]} for x in list(DB_CUR.execute('SELECT userid
    , text, date_day, date_month, date_year, positive_feeling FROM
    tweets'))]}}
```

No exemplo acima, um JSON com um atributo de erro e resposta é retornado. Dentro da resposta, há informações úteis para o frontend.

5.1. CSV

Também é possível baixar os tweets e os usuários através de CSV. Para tal, o método inlining também foi utilizado, e uma biblioteca que cria o CSV foi utilizada. Depois de criado, ele é enviado via API Rest normalmente.

```
@APP.route('/csv/all_tweets', methods=['GET'])
def csv__all_tweets():
    DB_CUR = get_database().cursor()
    dest = io.StringIO()
    writer = csv.writer(dest)
    writer.writerow(['text', 'date_day', 'date_month', 'date_year', '
    positive_feeling', 'userid', 'username', 'followers'])

    for i in list(DB_CUR.execute('SELECT text, date_day, date_month,
    date_year, positive_feeling, tweets.userid, username, followers
    FROM tweets INNER JOIN pages ON tweets.userid = pages.userid')):
```

```
        writer.writerow(i)

    output = make_response(dest.getvalue())
    output.headers["Content-Disposition"] = "attachment; filename=
    bubblegun_all_tweets.csv"
    output.headers["Content-type"] = "text/csv"
    return output
```

5.2. JSON

Além disso, é possível baixar os mesmos dados em JSON. Foi utilizada uma biblioteca que cria o JSON, e o arquivo é enviado via API Rest.

```
@APP.route('/json/all_pages', methods=['GET'])
def json__all_pages():
    DB_CUR = get_database().cursor()
    raw_json = json.dumps([{'userid': x[0], 'username': x[1], '
    followers': x[2]} for x in list(DB_CUR.execute('SELECT userid,
    username, followers FROM pages'))])

    output = make_response(raw_json)
    output.headers["Content-Disposition"] = "attachment; filename=
    bubblegun_all_pages.json"
    output.headers["Content-type"] = "text/json"
    return output
```

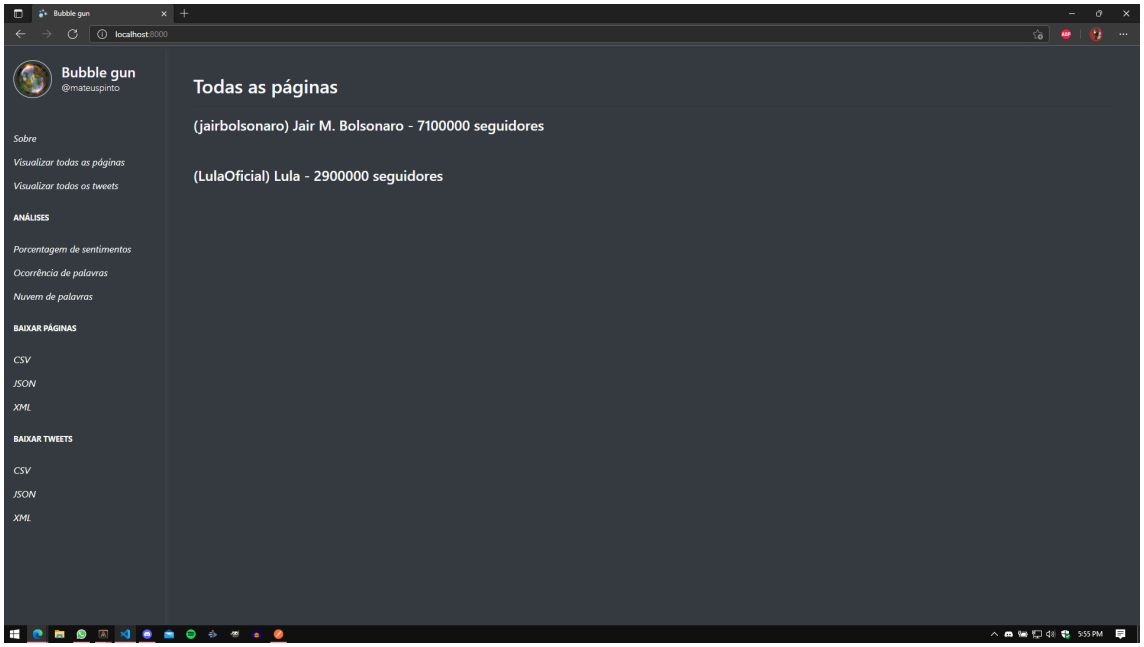
5.3. XML

Por último, é possível baixar tais informações via XML. Foi utilizada uma biblioteca externa que converte um dicionário em XML, e o arquivo então é enviado via API Rest.

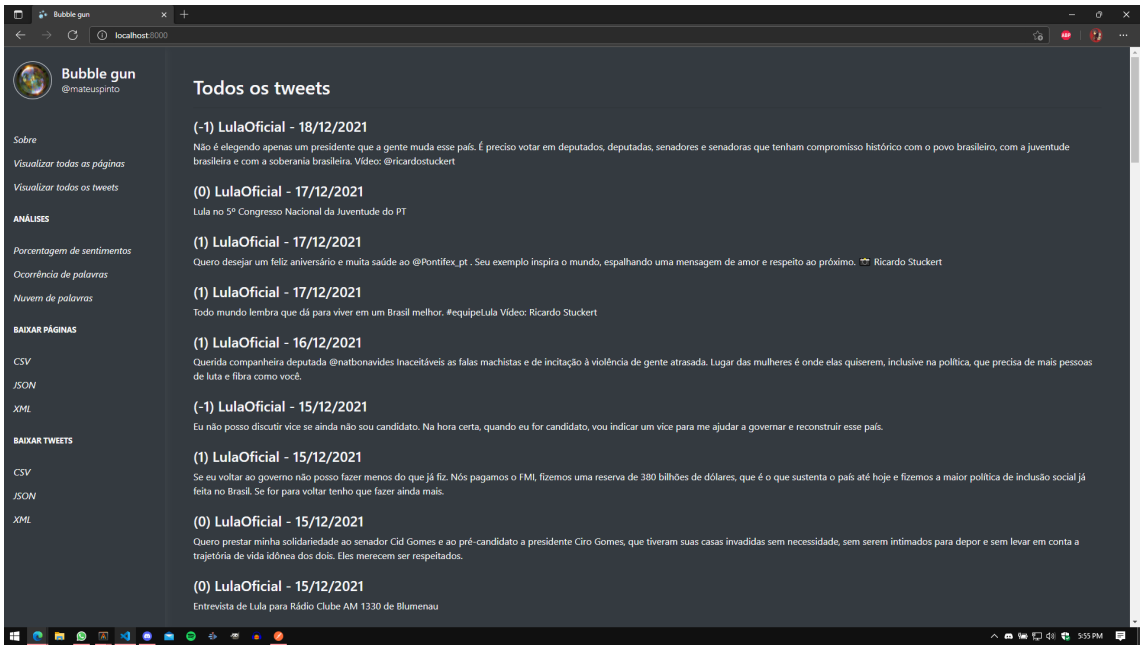
```
@APP.route('/xml/all_tweets', methods=['GET'])
def xml__all_tweets():
    DB_CUR = get_database().cursor()
    raw_xml = dicttoxml([{'text': x[0], 'date_day':x[1], 'date_month':x
    [2], 'date_year':x[3], 'positive_feeling':x[4], 'userid':x[5], '
    username':x[6], 'followers':x[7]} for x in list(DB_CUR.execute('
    SELECT text, date_day, date_month, date_year, positive_feeling,
    tweets.userid, username, followers FROM tweets INNER JOIN pages ON
    tweets.userid = pages.userid'))], custom_root='tweets', attr_type=
    False)

    output = make_response(raw_xml)
    output.headers["Content-Disposition"] = "attachment; filename=
    bubblegun_all_tweets.xml"
    output.headers["Content-type"] = "text/xml"
    return output
```

6. As funções do sistema

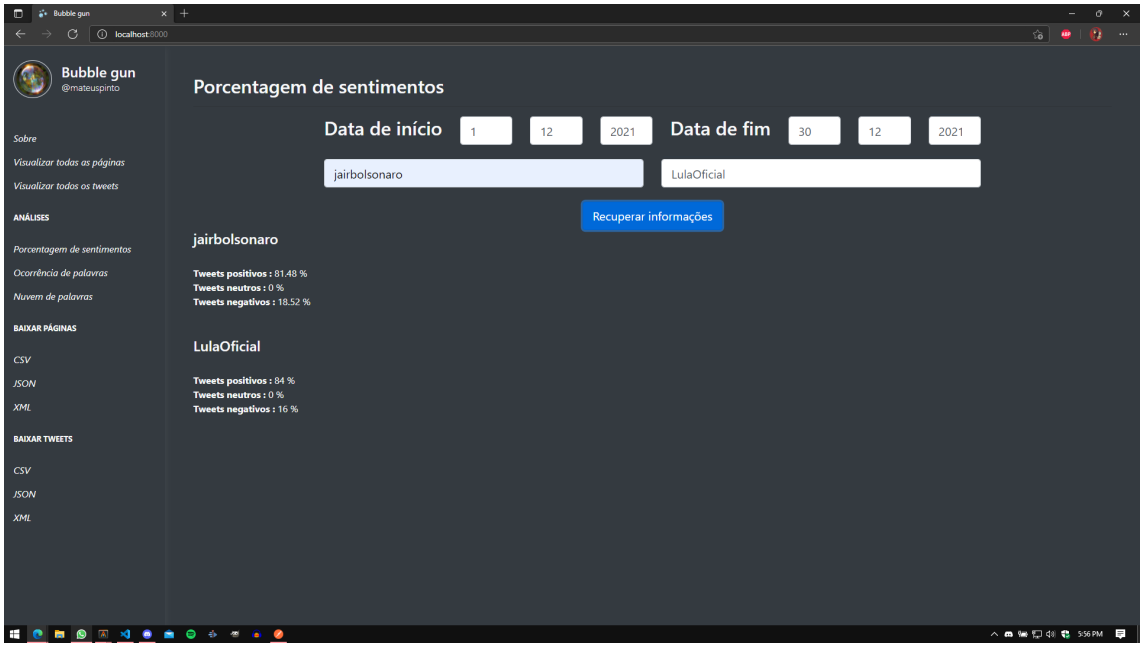
Todas as funções do sistema são focados em visualização e nos tweets. É possível ver quais usuários estão disponíveis no sistema.



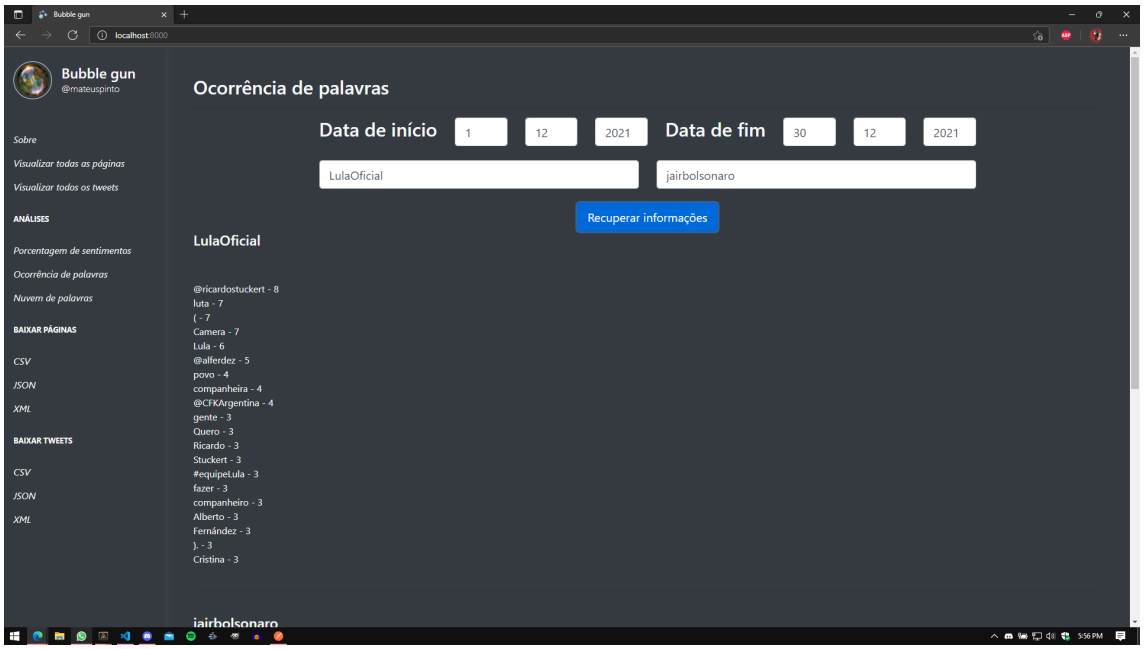
É possível ver todos os tweets cadastrados no sistema.



É possível ver as porcentagens de tweets positivos, neutros e negativos.



É possível ver as palavras mais recorrentes de cada candidato.



É possível ver uma nuvem de palavras de cada candidato.

7. Conclusão

Acredito que o trabalho tenha sido bastante produtivo no sentido de aprender mais sobre dados semiestruturados, embora o tempo tenha sido demasiadamente curto. Eu já tive experiências de ler JSON e CSV, porém escrevê-los foi algo bastante novo. Além disso, foi meu primeiro contato com XML. Gostei bastante de lidar com dados semiestruturados, embora eu ainda tenho receio em usar bancos NoSQL pela falta de consistência.