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

Luciana Farias de Oliveira Nóbrega

UM ESTUDO SOBRE DETECÇÃO DE DISCURSOS OFENSIVOS EM COMENTÁRIOS NA INTERNET UTILIZANDO APRENDIZADO DE MÁQUINA

Belo Horizonte
2021

Luciana Farias de Oliveira Nóbrega

UM ESTUDO SOBRE DETECÇÃO DE DISCURSOS OFENSIVOS EM COMENTÁRIOS NA INTERNET UTILIZANDO APRENDIZADO DE MÁQUINA

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
2021

SUMÁRIO

1. Introdução

1.1. Contextualização

Em julho de 2020, houve um grande boicote ao Facebook, com a alegação de que não havia esforço e trabalho suficiente para remover conteúdo de ódio e racista da rede social. A campanha contou com a adesão de pelo menos 240 marcas no mundo – entre elas estavam a Coca Cola, Volkswagen e Microsoft (REDAÇÃO GQ GLOBO, 2020).

Mais recentemente, o banco Bradesco iniciou uma campanha contra o assédio e importunação sexual, alterando as respostas da BIA (inteligência artificial do banco que faz atendimento aos clientes) quando algum discurso ofensivo vindo dos clientes é detectado, de forma a reagir de modo mais firme contra o assédio (BRADESCO, 2021).

As redes sociais apresentam atualmente um espaço de livre expressão para todas as pessoas, além de serem um espaço altamente explorado comercialmente pelas empresas como forma de divulgação de seus produtos e serviços. Nesse contexto, aliado ao crescimento do interesse em preservar a ética e o respeito a todos os seres humanos, um campo de pesquisa está em alta nos últimos anos: a utilização do aprendizado de máquina na detecção de discursos ofensivos em redes sociais e na *internet* de uma forma geral, sendo um desafio na área de Processamento de Linguagem Natural (PLN).

Este trabalho visa explorar, em um contexto de aprendizado de máquina, bases de dados com mensagens em português retiradas da *internet* e já classificadas como ofensivas ou não.

1.2. O problema proposto

Para melhor contextualizar o problema proposto, optou-se por utilizar a técnica dos 5-Ws, que propõem resumir o problema em cinco pontos principais: *who* (quem), *what* (o que), *when* (quando), *where* (onde) e *why* (por quê).

***Why*: Por que esse problema é importante?**

O problema proposto é importante em uma escala ética (discurso ofensivo é considerado ruim eticamente) e comercial (como exemplo, tem-se o caso do com a campanha *Stop Hate for Profit*, ou “Pare o ódio por lucro”, em tradução livre, que, entre outras coisas, fez o Facebook perder patrocinadores por não fazer um bom tratamento de discursos ofensivos publicados na plataforma).

***Who*: De quem são os dados analisados?**

Os dados analisados foram retirados da internet. Duas bases de dados foram utilizadas para a classificação: um retirado diretamente da rede social Twitter (FORTUNA, 2019), e outro de comentários em notícias no site jornalístico G1 (PELLE, 2017).

***What*: Quais os objetivos com essa análise?**

Serão analisados alguns padrões de linguagem que caracterizam discursos ofensivos, de forma a analisar se determinada frase ou comentário é ou não ofensivo. Com esses padrões identificados, é possível gerar um modelo que pode detectar automaticamente esse tipo de discurso na *internet*, de forma que eles podem ser localizados mais facilmente, e cada plataforma pode agir de acordo com seus termos de uso: removendo ou ocultando o comentário, por exemplo.

***Where*: Quais são os aspectos geográficos e logísticos da análise?**

O estudo sobre análise de discursos ofensivos/discursos de ódio é amplamente discutido, devido à sua importância ética e comercial. Contudo, não há muitos estudos em língua portuguesa. Assim, o foco será em comentários publicados principalmente (mas não somente) em países lusófonos, como Brasil e Portugal.

***When*: Qual o período está sendo analisado?**

Os dados analisados foram obtidos com base em dois estudos, (PELLE, 2017) e (FORTUNA, 2019). Assim, os dados analisados são dos anos de 2017 e 2019.

2. Coleta de Dados

A página hatespeechdata.com fornece uma coleção de base de dados já classificados de discurso de ódio, abuso virtual e linguagem ofensiva em diversos idiomas. O propósito é justamente fornecer *datasets* para todos aqueles que desejarem fazer o processamento de linguagem natural para identificar discursos ofensivos. Nessa plataforma, foram encontrados duas bases de dados criados a partir de dois diferentes estudos.

PELLE (2017) recolheu os dados a partir do site de notícias G1. Apesar da plataforma ser monitorada, um alto número de comentários ofensivos foi localizado, especialmente em notícias de esportes e política. Assim, a coleta de dados foi limitada a essas duas sessões, resultando em uma base de dados de 10.336 comentários, retirados de 115 notícias. Porém, os comentários deveriam ser classificados por humanos e, como seria inviável classificar manualmente uma quantidade grande de dados, foram escolhidos aleatoriamente alguns comentários para serem classificados manualmente por três pessoas. Com isso, duas bases de dados foram geradas: OFFCOMBR-2, quando duas pessoas classificaram o discurso como ofensivo, e OFFCOMBR-3, quando todos os três classificadores assim o julgaram. Para efeitos deste trabalho, iremos usar apenas o OFFCOMBR-3, por acreditar que os padrões de discurso ofensivos estejam mais evidentes, facilitando o trabalho de classificação. Essa base de dados possui as seguintes propriedades:

Tabela 1: Informações sobre o conjunto de dados retirado de (PELLE, 2017).

Nome da coluna/campo	Descrição	Tipo
Class	Identificação sobre se o texto foi classificado como discurso ofensivo (valor *yes*) ou não (valor *no*)	*String*
data	Texto publicado pelo usuário no *site* de notícias G1	*String*

Além disso, há um total de 1.033 comentários, sendo 831 considerados não ofensivos, enquanto 202 foram classificados assim.

O *dataset* foi obtido diretamente do repositório do GitHub do projeto, salvo no arquivo dataset_depelle_3.csv e esse arquivo foi lido de forma a gerar um Pandas DataFrame, conforme apresentado abaixo:

```
url = "https://raw.githubusercontent.com/rogersdepelle/OffComBR/master/Off-
ComBR3.arff"
f = urlopen(url)
dataset_depelle_3 = f.read()
f.close()

csv_file = open('datasets/dataset_depelle_3.csv', 'wb')
csv_file.write(dataset_depelle_3)
csv_file.close()
dataset_depelle_3 = pd.read_csv("datasets/dataset_depelle_3.csv")
```

O segundo conjunto de dados utilizado foi desenvolvido no trabalho (FORTUNA, 2019). Os dados foram coletados na plataforma Twitter, com foco em palavras-chave e *hashtags* relacionadas ao discurso de ódio como, por exemplo, #LugarDeMulherÉNaCozinha. Após a coleta de dados, os *tweets* foram classificados manualmente com anotações binárias: ofensivo/não-ofensivo. Em seguida, especialistas no assunto classificaram o discurso ofensivo em 81 categorias distintas (sexismo, racismo, homofobia, religião etc.). Para efeitos deste estudo, será considerada apenas a classificação binária de discurso ofensivo. O conjunto de dados tem um total de 5.668 comentários do Twitter, sendo 4.440 classificados como não-ofensivo, e 1.228 como ofensivo. Algumas propriedades desse *dataset* são apresentadas abaixo. A classificação específica do discurso de ódio em categorias foi feita em formato *dummy*, ou seja, cada categoria é representada por uma coluna, e cada linha terá o valor de 0 ou 1 caso seja classificado com determinada categoria.

Tabela 2: Informações sobre o conjunto de dados retirado de (FORTUNA, 2019).

Nome da coluna/campo	Descrição	Tipo
Text	*Tweet* a ser classificado	*String*
Hate.speech	Identificação sobre se o texto foi classificado como discurso ofensivo (valor 1) ou não (valor 0)	*Bool*
Sexism	Identificação sobre se o texto foi classificado como discurso sexista (valor 1) ou não (valor 0)	*Bool*
Body	Identificação sobre se o texto foi classificado como discurso ofensivo em relação ao corpo de outrem (valor 1) ou não (valor 0)	*Bool*
...	...	...

A coleta desse conjunto de dados foi realizada de forma muito similar ao OFFCOMBR-3, de modo que os dados foram retirados diretamente do GitHub do projeto, armazenados em um arquivo csv chamado dataset_fortuna.csv e exportado em um Pandas DataFrame.

```
url = "https://raw.githubusercontent.com/paulafortuna/Portuguese-Hate-Spe-
ech-Dataset/master/2019-05-28_portuguese_hate_speech_hierarchical_classifi-
cation.csv"
f = urlopen(url)
dataset_fortuna = f.read()
f.close()

csv_file = open('datasets/dataset_fortuna.csv', 'wb')
csv_file.write(dataset_fortuna)
csv_file.close()

dataset_fortuna = pd.read_csv("datasets/dataset_fortuna.csv")
```

De posse de dois conjuntos de dados diferentes, foi necessário manipulá-los com o objetivo de padronizá-los e ser possível realizar sua concatenação, de forma a

obter um único *dataset*. Decidiu-se modificar os conjuntos de dados para apresentar as seguintes informações:

Tabela 3: Informação sobre o conjunto de dados a ser utilizado neste trabalho.

Nome da coluna/campo	**Descrição**	**Tipo**
text	Texto a ser analisado	*String*
hate_speech	Identificação sobre se o texto foi classificado como discurso ofensivo (valor 1) ou não (valor 0)	*Bool*

Inicialmente, os dados *dataset_depelle_3* foram alterados revertendo as colunas (para apresentar na ordem desejada), alterando a classificação de yes/no para 1/0 e, por fim, renomeando as colunas para *text* e *hate_speech.* A forma como foi feita essa modificação é apresentada abaixo:

```
cols = dataset_depelle_3.columns.tolist()
cols = cols[::-1] #reverting cols
dataset_depelle_3 = dataset_depelle_3[cols]
dataset_depelle_3['class'] = dataset_depelle_3['class'].map(
                                                    {'yes': 1,
                                                     'no': 0})
dataset_depelle_3.columns = ['text', 'hate_speech']
```

Para o conjunto de dados *dataset_fortuna*, foi necessário apenas remover as colunas de classificação específica, e renomear o campo *Hate.speech* para *hate_speech*, conforme feito abaixo:

```
dataset_fortuna = dataset_fortuna[['text', 'Hate.speech']]
dataset_fortuna.columns = ['text', 'hate_speech']
```

Por fim, foi necessário concatenar os dados utilizando a função *concat*:

```
full_dataset3 = pd.concat([dataset_fortuna, dataset_depelle_3],
                          ignore_index=True)
```

O conjunto de dados final possui 6.701 comentários, sendo 5.271 considerados não-ofensivo e 1.430 ofensivos.

3. Processamento/Tratamento de Dados

Em seguida, será necessário realizar o processamento e tratamento dos dados. Logo no início do processo, uma vantagem do preenchimento de dados de forma manual foi percebida: apesar de ser uma tarefa maçante, não foram encontrados valores nulos ou ausentes. Contudo, o processamento de linguagem natural envolve diversos passos, explanados a seguir.

3.1 Tratamento de dados desbalanceados

Como falado anteriormente, os dados estão claramente desbalanceados. Para melhor visualização, plotamos um gráfico com a distribuição de dados entre as classes, criando a função *plot_count* e utilizando as bibliotecas *matplotlib* e *Counter*.

```
import matplotlib.pyplot as plt
from collections import Counter

def plot_count(count):
    plt.rcdefaults()
    fig, ax = plt.subplots()
    count0 = counter[0]
    count1 = counter[1]
    total = count0 + count1
    amount = (count0/total, count1/total)
    ax.barh([0,1], amount, align='center')
    ax.set_yticks([0,1])
    ax.set_yticklabels([0,1])
    ax.invert_yaxis()  # labels read top-to-bottom
    ax.set_xlabel('% of data')
    ax.set_ylabel('Hate detection')
    ax.set_title('Data distribution')
    a = round(amount[0], 3)
    ax.text(a, 0, str(a*100) + '%', color='black')
    b = round(amount[1], 3)
    ax.text(b, 1, str(b*100) + '%', color='black')
    plt.xlim([0.0, 1.0])
    plt.show()
counter = Counter(full_dataset3['hate_speech'])
plot_count(counter)
```

Com a Figura 1, nota-se mais facilmente que há uma predominância da classe 0, que indica que não há discurso ofensivo. No conjunto de dados analisados, 78.7% foram classificados como não-ofensivos, enquanto 21.3% foram considerados ofensivos. Desse modo, decidiu-se por alternativas para a análise do problema, visto que um conjunto de dados desbalanceado pode enviesar o modelo a classificar os dados com a classe dominante.

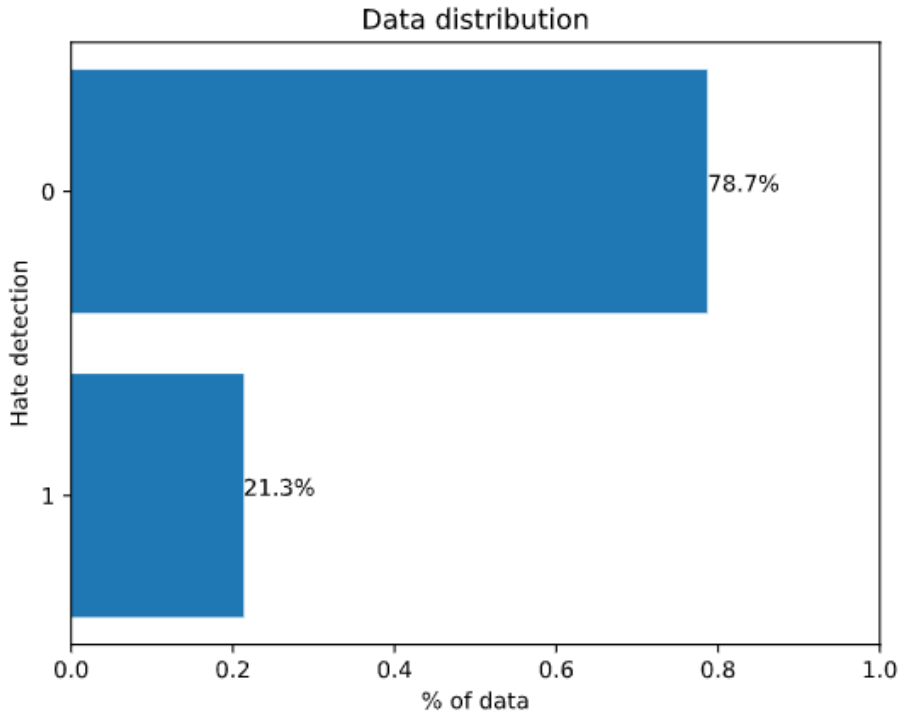

Figura 1: Distribuição do conjunto de dados entre as classes.

Três técnicas serão analisadas: subamostragem, ou *undersampling*, e sobreamostragem, ou *oversampling*, detalhadas a seguir. Após finalizar essa etapa, serão obtidos os quatro conjuntos de dados a serem comparados neste trabalho, nomeados da seguinte maneira:

- *Unbalanced*: conjunto de dados desbalanceados;
- *Undersampling*: conjunto de dados que passaram pela subamostragem;
- *Repeating*: conjunto de dados que foram sobreamostrados por repetição;
- *Translate*: conjunto de dados que foram sobreamostrados com adição de um elemento aleatório (tradução).

3.1.1 Subamostragem aleatória

A técnica de subamostragem aleatória consiste em eliminar, de forma randomizada, elementos da classe majoritária, de forma a ter a mesma quantidade de elementos da classe minoritária. É um método simples, porém, pode aumentar a variância do classificador e há chances de descartar amostras úteis ou importantes, afetando a habilidade de generalização do classificador (FERNÁNDEZ, 2018).

A subamostragem aleatória foi implementada da maneira apresentada abaixo. A função *undersampling* recebe como parâmetros o *DataFrame* a ser subamostrado e o valor possível das classes (neste caso, 0 e 1). A função calcula a diferença entre a quantidade de amostras de cada classe e armazena na variável *diff*, separa as amostras de cada classe em *DataFrames* distintos, elimina *diff* amostras da classe majoritária e concatena as amostras da classe minoritária com a classe majoritária reduzida.

```
full_dataset3_unbalanced = full_dataset3.copy()

def undersampling(dataframe, class_values=[0,1]):
    count0 = dataframe[dataframe.hate_speech==class_values[0]].count()
    count1 = dataframe[dataframe.hate_speech==class_values[1]].count()

    df1 = dataframe[dataframe.hate_speech==class_values[1]]

    diff = abs(count0.text - count1.text)
    list_messages = dataframe[dataframe.hate_speech==class_val-
ues[0]]['text'].tolist()

    for j in range(diff):
        ind = random.randint(0, len(list_messages))
        del list_messages[ind-1]

    df = DataFrame({'text':list_messages, 'hate_speech': np.re-
peat(0, len(list_messages))})
    data = pd.concat([df1, df])
    return shuffle(data)

full_dataset3_balanced_undersampling =
                      undersampling(full_dataset3_unbalanced)
```

Após criar o DataFrame *full_dataset3_balanced_undersampling*, utilizamos a função Counter para checar a quantidade de dados de cada classe, comprovando que as classes foram balanceadas, e ambas possuem 1.430 elementos cada.

```
counter = Counter(full_dataset3_balanced_undersampling['hate_speech'])
print('Class 0: {}, Class 1: {}'.format(counter[0],counter[1]))
```

```
Class 0: 1430, Class 1: 1430
```

3.1.2 Sobreamostragem aleatória

Devidos aos problemas que podem decorrer devido à subamostragem, a técnica de sobreamostragem foi desenvolvida com base na ideia de replicar a classe minoritária, de forma que seja balanceada com a classe majoritária. No caso da sobreamostragem aleatória, elementos randomizados da classe minoritária serão replicados. A implementação é apresentada a seguir, com a função

```
def data_augmentation_repeating(dataframe, class_values=[0,1]):
    count0 = dataframe[dataframe.hate_speech==class_values[0]].count()
    count1 = dataframe[dataframe.hate_speech==class_values[1]].count()

    aug_range = abs(count0.text - count1.text)

    list_messages = dataframe[dataframe.hate_speech==class_val-
ues[1]]['text'].tolist()

    len_list = len(list_messages)
    augmented_messages = []

    for j in range(aug_range):
        ind = random.randint(0, len_list)
        message = list_messages[ind-1]

        augmented_messages.append([message, 1])

    df = DataFrame(augmented_messages)
    df.columns = ['text', 'hate_speech']

    data = pd.concat([dataframe, df])

    return shuffle(data)


full_dataset3_balanced_repeating = data_augmentation_repeating(full_da-
taset3_unbalanced)
```

data_augmentation_repeating e com a criação do *DataFrame* *fulldataset3_balanced_repeating*.

3.1.3 Sobreamostragem com adição de elementos aleatórios

A técnica de e sobreamostragem aleatória pode parecer interessante em um primeiro momento. Contudo, a simples adição de novos elementos, apenas implica em um maior peso ou custo para as classes minoritárias (FERNÁNDEZ, 2018). Além disso, a sobreamostragem aleatória, *a priori*, pode apenas enviesar o modelo, podendo ocasionar em *overfitting*, ou sobreajuste, isto é, quando o modelo classifica muito bem dados que já foram a ele apresentados, mas classifica de modo mais errôneo novos dados.

Desse modo, é interessante comparar o desempenho de uma sobreamostragem que replica a classe minoritária adicionando elementos aleatórios às amostras. Em (AROYEHUN, 2018), a adição de elementos aleatórios na sobreamostragem foi realizada a partir da tradução do texto para quatro idiomas intermediários (francês, espanhol, alemão e hindi), e então de volta para inglês. Nesse estudo, foi observado um aumento um aumento de performance no modelo. Assim, optou-se por utilizar essa técnica.

Neste trabalho, utilizou-se o método *translate* da biblioteca TextBlob para realizar a tradução. Essa biblioteca utiliza a API do Google Tradutor e, com isso, notou-se uma dificuldade na utilização desse método da tradução, pois essa API bloqueia a utilização quando há um alto número de solicitações HTTP em pouco tempo, de forma de seu uso é liberado novamente apenas quando a cota diária de requisições é redefinida, sempre à meia-noite no horário do Pacífico (GOOGLE CLOUD, 2020). Uma opção encontrada foi a de adicionar um tempo de dois segundos entre uma requisição e outra. Apesar do longo tempo de processamento, foi possível realizar a tradução dos textos da classe minoritária. Para evitar o retrabalho de processar todos os dados novamente, o *DataFrame* *full_dataset3_balanced_translated* foi armazenado em formato csv, para ser reutilizado sempre que necessário.

No trecho de código abaixo, é apresentada a função *data_augmentation_translate*:

```
from textblob import TextBlob
from textblob.translate import NotTranslated
import random
from tqdm import tqdm
import time
language = ["fr", "la", "es", "de", "it", "en", "ru", "zh", "fi"]
sr = random.SystemRandom()
def data_augmentation_translate(dataframe, class_values=[0,1]):
    count0 = dataframe[dataframe.hate_speech==class_values[0]].count()
    count1 = dataframe[dataframe.hate_speech==class_values[1]].count()
    aug_range = abs(count0.text - count1.text)
    list_messages = dataframe[dataframe.hate_speech==class_val-
ues[1]]['text'].tolist()
    len_list = len(list_messages)
    augmented_messages = []
    for j in tqdm(range(0, aug_range)):
        ind = random.randint(0, len_list)
        message = list_messages[ind-1]
        if hasattr(message, "decode"):
            message = message.decode("utf-8")
        text = TextBlob(message)
        try:
            text = text.translate(to=sr.choice(language))
            time.sleep(2)
            text = text.translate(to="pt")
        except NotTranslated:
            pass
        augmented_messages.append([str(text), 1])
    df = DataFrame(augmented_messages)
    df.columns = ['text', 'hate_speech']

    data = pd.concat([dataframe, df])
    return shuffle(data)
try:
    full_dataset3_balanced_translate = pd.read_csv('datasets\\full_da-
taset3_balanced_translate.csv')
except FileNotFoundError:
    full_dataset3_balanced_translate = data_augmentation_translate(full_da-
taset3_unbalanced)
    full_dataset3_balanced_translate.to_csv('datasets\\full_dataset3_bal-
anced_translate.csv', index=False)
```

3.2 Pré-processamento do texto

Nesta subseção, serão apresentadas as técnicas de pré-processamento de texto utilizadas neste trabalho, e que são comumente utilizadas em projetos de Processamento de Linguagem Natural (PLN).

3.2.1 Remoção de caracteres desnecessários

A análise léxica neste trabalho consistiu em converter todo o texto para caracteres minúsculos, remover *links*, números, pontuações, marcações de outros usuários (representadas pelo símbolo arroba, "@"), e remover a expressão "RT" (oriunda do termo *retweet*, que indica que o texto fora apenas copiado por outro usuário). Contudo, alguns caracteres e palavras foram julgadas como essenciais para a diferenciação de um discurso ofensivo, e optou-se por não as remover. A palavra "não" foi transformada em uma marcação "*_NAO*", e os sinais de interrogação e exclamação foram substituídos por "*_Q*" e "*_E*", respectivamente.

Para essa tarefa, foram utilizadas expressões regulares, conforme apresentado no trecho de código abaixo:

```
import re

def cleaning_data(text):
    text = re.sub(r'[0-9]', '', text)
    text = text.lower()
    text = re.sub(r'nao', '_NAO', text)
    text = re.sub(r'não', '_NAO', text)
    text = text.replace('?', '_Q')
    text = text.replace('!', '_E')

    text = re.sub(r'http\S+', '', text)
    text = re.sub(r'www\S+', '', text)
    text = re.sub(r'rt+', '', text)
    text = re.sub(r'@\w+', '', text)
    text = re.sub(r'[\"!$%-*&’‘*+”“,-./\[\]<=>\'?():;^_`<>{|}~@]', '', text)
    return text
```

3.2.2 Remoção de *stopwords*

As *stopwords* são palavras frequentes, que não possuem relevância para o entendimento de uma sentença e, portanto, podem ser simplesmente ignoradas, sem afetar a qualidade do modelo.

Como é de se esperar, cada idioma possui seus respectivos *stopwords* e, para obter as palavras pouco relevantes em português, foi utilizada a biblioteca nltk (*Natural Language Toolkit*). Assim, foi gerado um conjunto de 204 *stopwords* no idioma português, que inclui palavras como preposições (de, com, como, para, entre, etc), artigos (a, o, uma, um, etc), pronomes (eu, ele, aquele, meu, etc), alguns verbos de ligação (ser, estar, haver, for, etc), entre outras. A maneira que a remoção de *stopwords* foi realizada é apresentada abaixo, na função *remove_stopwords.* É importante notar que alguns elementos do conjunto de dados eram compostos por *stopwords*, apenas (por exemplo: “quando foi isso”). Para esses casos, foi decidido por manter o texto original.

```
import nltk

# get stop words in portuguese
stopwords = set(nltk.corpus.stopwords.words('portuguese'))

def remove_stopwords(text):
    text_without_stop-
words = [word for word in text.split() if word not in stopwords]
    if text_without_stopwords == []:
        return text
    return ' '.join(text_without_stopwords)
```

3.2.3 Remoção de radicais: a stemização

A stemização (do inglês, *stemming*) é o processo de reduzir palavras, com o objetivo de reduzir o tamanho da estrutura de indexação ou, em outras palavras, reduzir o número de palavras distintas. Em português, palavras como verbos e substantivos possuem diversas variações, e a stemização irá padronizar essas palavras. Como exemplo, tem-se a palavra “casa” e suas variantes: “casas”, “casinha”, “casinhas”, “casarão”, “casarões”. Após a remoção de radicais, todas essas palavras serão transformadas em “cas”. Um outro exemplo é o verbo “andar”: variações como “andei”, “andamos”, “andarão”, se tornarão “and”.

Para a implementação, foi necessário utilizar o *stem.RSLPStemmer* da biblioteca nltk. Como esse método funciona apenas para palavras separadas, foi necessário entrar com a frase completa, separar as palavras pelo método *split*, processar cada

palavra em um laço de repetição e juntar cada palavra pelo método *join*, conforme apresentado no trecho de código abaixo:

```
import nltk
stem = nltk.stem.RSLPStemmer()

def stemming(text):
    text = text.split()
    new_text = []
    for word in text:
        new_text.append(stem.stem(word))
    return ' '.join(new_text)
```

3.2.4 Seleção de termos

Em seguida, foi decido por etiquetar (*tag*) determinadas palavras que podem ser padronizadas e utilizadas da mesma maneira pelo modelo. Essas palavras foram as de baixo calão (*swear words*) e as risadas.

Para identificar as palavras de baixo calão, foi utilizado o conjunto de dados presentes em (RANZI, 2020), que é uma lista de diversos palavrões que pode ser utilizada para, por exemplo, identificar e bloquear essas palavras em um *blog* pessoal ou mesmo em um *site* comercial. De posse dessa lista, foi possível processar cada frase e substituir os palavrões pela tag *_SWEARWORD*. Isso é representado abaixo, na função *tagging_swarwords*.

```
f = open("datasets\\lista-palavroes-bloqueio.txt", "r", encoding="utf8")
list_bad_words = f.readlines()
list_bad_words = [x.strip() for x in list_bad_words]
f.close()

def tagging_swearwords(text):
    text = [word if (word.upper() not in list_bad_words) else "_SWEAR-
WORD" for word in text.split()]
    text = ' '.join(text)
    return text
```

Além disso, foi decido padronizar as risadas deixadas pelo usuário. As risadas foram identificadas empiricamente, criando um padrão, identificado por expressões

regulares e substituído pela tag *_LAUGHS*, conforme observado no trecho de código abaixo, na função *tagging_laughs*. As risadas identificadas foram as que seguem um padrão como “kkkkk”, “haha”, “hehe” e “rsrs”, empiricamente visto como as mais comuns no conjunto de dados analisado.

```
def tagging_laughs(text):
    return  re.sub(r'k{2,}|a*ha+h[ha]*|e*he+h[he]*|s*rs+r[rs]*',
                   '_LAUGHS ',
                   text)
```

3.2.5 Criação de diferentes métodos de pré-processamento

Para efeitos de estudo da influência de cada etapa do pré-processamento de texto, decidiu-se criar quatro sequências, que serão comparadas mais à frente.

Primeiramente, há o método *preprocessing1*, que inclui a limpeza de dados e remoção de *stopwords* (conforme apresentado nas subseções 3.2.1 e 3.2.2). Além disso, adicionou-se mais um passo: a remoção de acentuação nas palavras. Optou-se pela inclusão desse passo, pois, na Internet, é comum a falta de acentuação nas palavras e, ao remover de todas as palavras, é possível ter uma melhor padronização. Por fim, todo o texto é dividido em palavras (método *split* do Python). A remoção de acentos e a divisão em palavras é também adicionada a todos os próximos métodos de pré-processamento.

```
from unidecode import unidecode

def preprocessing1(text):
    text = cleaning_data(text)
    text = remove_stopwords(text)
    text = unidecode(text) #remove accents (e.g.: á, ã, à will be "a")
    return text.split()
```

O segundo método de pré-processamento é o *preprocessing2*, que é semelhante ao anterior, mas adicionando o *stemming*:

```
def preprocessing2(text):
    text = cleaning_data(text)
    text = remove_stopwords(text)
    text = stemming(text)
    text = unidecode(text)
    return text.split()
```

As funções a seguir, *preprocessing3* e *preprocessing4* são semelhantes aos anteriores, mas adicionam a seleção de termos, conforme definido em 3.2.4.

```
def preprocessing3(text):
    text = cleaning_data(text)
    text = remove_stopwords(text)
    text = unidecode(text)
    text = tagging_swearwords(text)
    text = tagging_laughs(text)
    return text.split()

def preprocessing4(text):
    text = cleaning_data(text)
    text = remove_stopwords(text)
    text = stemming(text)
    text = unidecode(text)
    text = tagging_swearwords(text)
    text = tagging_laughs(text)
    return text.split()
```

Resumindo todos os métodos apresentados acima, há a Tabela 4, abaixo:

Tabela 4: Resumo das funções de pré-processamento.

	cleaning_data	*remove_stopwords*	*stemming*	*tagging*
preprocessing1	×	×		
preprocessing2	×	×	×	
preprocessing3	×	×		×
preprocessing4	×	×	×	×

3.3 Vetorização

Com os métodos de pré-processamentos criados, o próximo passo é converter cada uma das mensagens em um vetor de número inteiros ou de pontos flutuantes que os modelos de aprendizado de máquina do *Scikit-learn* possam trabalhar.

3.3.1 Criação do modelo *bag of words*

O modelo *bag of words* (saco de palavras) irá contar quantas vezes uma palavra ocorre em cada mensagem. O *CountVectorizer* é usado para converter uma coleção de textos em um vetor de contagem de termos. Ele também permite o pré-processamento dos dados de texto antes de gerar a representação vetorial.

O resultado do processamento com o *CountVectorizer* irá gerar uma matriz de duas dimensões, na qual a primeira é todo o vocabulário, isto é, o conjunto de todas as palavras usadas em todas as mensagens (uma palavra por linha) e a outra dimensão é de fato os documentos, ou seja, uma coluna por mensagem. Visto que existe muitas mensagens, é esperado que seja obtido uma matriz esparsa de altas dimensões.

Para efeitos comparativos, processamos cada um dos conjuntos de dados utilizando cada um dos pré-processadores criados.

Tabela 5: Dimensões das matrizes esparsas geradas pela vetorização (coluna x linha).

	unbalanced	*undersampling*	*repeating*	*translate*
preprocessing1	6701x13753	2860x7937	10542x13753	10542x16050
preprocessing2	6701x8095	2860x5080	10542x8095	10542x9367
preprocessing3	6701x13586	2860x7806	10542x13586	10542x15871
preprocessing4	6701x8023	2860x5029	10542x8023	10542x9286

Nota-se que, de fato, a matriz criada possui a mesma quantidade de linhas de cada coluna representa uma mensagem, enquanto a quantidade de linhas varia de acordo com o método de pré-processamento escolhido. Além disso, percebe-se que:

- A stemização reduziu, em média, 40% a quantidade de palavras (comparação dos métodos *preprocessing1* com *preprocessing2* e *preprocessing3* com *preprocessing4)*;
- A seleção de palavras de baixo calão e risadas reduziu em média 1,1% a quantidade de palavras (comparação do *preprocessing1* com *preprocessing3* e *preprocessing2* com *preprocessing4*);
- A utilização do método de tradução para adicionar variabilidade aos dados se mostrou efetiva: houve aumento de 14% no número de palavras (comparação dos conjuntos de dados *repeating* e *translate*).

3.3.2 TF-IDF

TF-IDF é uma sigla em inglês que significa *Term Frequency-Inverse Document Frequency*, isto é, a frequência do termo–inverso da frequência nos documentos. Ele tenta expressar a importância e frequência de uma palavra dentro de uma coleção de textos.

Normalmente, o peso TF-IDF é composto por duas partes: o primeiro (TF) calcula a frequência do termo normalizada, isto é, o número de vezes que uma palavra aparece em um documento, dividido pela quantidade total de palavras naquele documento; já o segundo, IDF, calcula o logaritmo do número de documentos no conjunto, dividido pelo número de documentos onde determinada palavra aparece. Assim, a importância de uma palavra aumenta quanto mais rara ela for. O TF-IDF é calculado como apresentado na Equação (1).

$$tfidf(t,d,D) = \frac{f_{t,d}}{\sum_{t' \epsilon\, d} f_{t',d}} \cdot log \frac{N}{|\{d \,\epsilon\, D : t \,\varepsilon\, d\}|} \quad (1)$$

Onde:

- $f_{t,d}$ é o número de vezes que o termo *t* aparece no documento *d*;
- $\sum_{t' \epsilon\, d} f_{t',d}$ é o total de termos no texto;
- N é o número total de documentos de uma coleção;
- $|\{d \,\epsilon\, D : t \,\varepsilon\, d\}|$ é o número de documentos onde o termo *t* ocorreu.

4. Análise e Exploração dos Dados

Nessa seção, serão analisadas algumas estatísticas dos dados obtidos, que contém apenas um texto e um classificador de discurso ofensivo ou não.

O objetivo aqui é explorar se há relação entre algumas características do texto (comprimento do texto, quantidade de palavras soletradas de forma errada e presença de palavras de baixo calão) e o fato do texto ser considerado ofensivo ou não. Para essa análise, o conjunto de dados *unbalanced* foi utilizado.

Além disso, diferentes pré-processamentos, específicos para a análise exploratória dos dados, foram utilizados: um para apenas remover *links* (para auxiliar na análise do comprimento do texto) e outro para remover links e eliminar caracteres indesejados (na análise de palavras escritas fora do padrão da norma culta). Esse processo foi utilizado para remover apenas as informações que não são importantes para a análise específica.

```
def removing_links(text):
    text = text.lower()
    text = re.sub(r'http\S+', '', text)
    text = re.sub(r'www\S+', '', text)
    return text

def cleaning_data(text):
    text = re.sub(r'[0-9]', '', text)
    text = re.sub(r'rt+', '', text)
    text = re.sub(r'@\w+', '', text)
    text = re.sub(r'#\w+', '', text)
    text = re.sub(r'[\"!$%-*&’‘*+”“,-./\[\]<=>\'?():;^_`<>{|}~#@]', '',
text)
    return text

def preprocessing(text):
    text = removing_links(text)
    text = cleaning_data(text)
    return text
```

Com isso, modificar-se-á o texto duas vezes, e esses conjuntos serão salvos no mesmo *DataFrame.* No trecho de código abaixo, importa-se a base de dados original (texto presente na coluna *text*), modifica-o para remover os links, armazena o novo conjunto de dados na coluna *text_removing_links* e faz o pré-processamento completo, armazenando na coluna *text_preproc*.

```
full_dataset3 = pd.read_csv('datasets\\full_dataset_3.csv')
full_dataset3['text_removing_links'] =
              full_dataset3['text'].apply(removing_links)
full_dataset3['text_preproc] = full_dataset3['text'].apply(preprocessing)
```

4.1 Tamanho do texto

Para observar a influência do tamanho do texto, utilizou-se a biblioteca *seaborn* para plotar gráficos do tipo *boxplot* das duas classes analisadas. Para calcular o comprimento do texto, utilizou-se o método *len* do Python.

Conforme explanado anteriormente, o comprimento dos *links* adicionados no texto será ignorado, pois eles não adicionam informações que indicam ou não o discurso ofensivo.

O trecho de código utilizado e o *plot* da imagem são apresentados abaixo:

```
import seaborn as sns
full_dataset3['length'] = full_dataset3['text_removing_links'].apply(len)
sns.boxplot(x='hate_speech', y='length', data=full_dataset3)
```

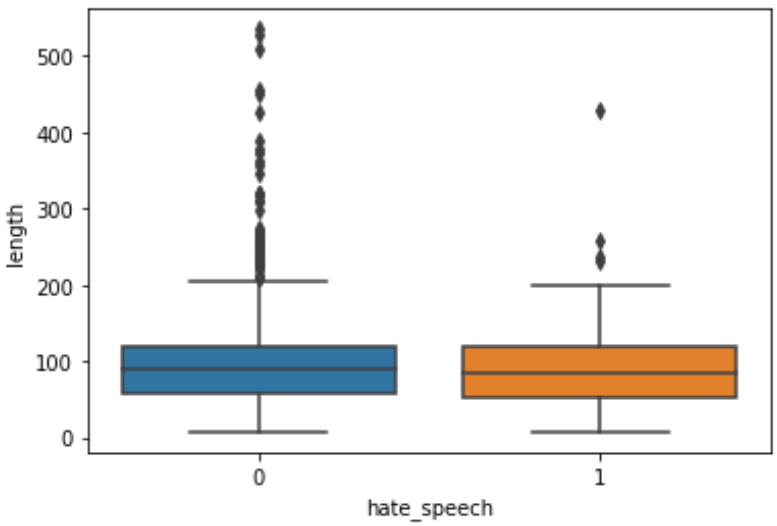

Figura 2: Comparação de tamanho de texto entre as classes.

Ao analisar o gráfico, nota-se que a mediana de ambas as classes é similar. A classe 0 (discurso não ofensivo) tem mediana de 90 caracteres, enquanto a classe 1 o valor é de 84, com menos valores discrepantes (*outliers*).

Notou-se também que a mediana do comprimento de todos os elementos do conjunto de dados é de 88 caracteres, e que 50,73% dos elementos da classe 0 estão acima desse valor, e na classe 1, 47,06% são acima. Conclui-se que os discursos considerados não ofensivos são ligeiramente maiores (com mais caracteres) que os ofensivos.

4.2 Quantidade de palavras soletradas erroneamente

Para identificar uma palavra escrita na forma fora da norma culta da língua, utilizou-se a biblioteca *SpellChecker*. Com ela, criou-se a função *count_misspell*, que tem como entrada um texto e, como saída, a quantidade de palavras soletradas de forma incorreta, verificado com o *SpellChecker*. O texto utilizado para realizar a contagem de palavras incorretas, conforme explicado anteriormente, foi o texto *text_*preproc, em que houve remoção de *links*, pontuação, *hashtags* e identificadores de usuários, conforme apresentado no trecho de código a seguir:

```
from spellchecker import SpellChecker
spell = SpellChecker(language='pt')

def count_misspell(text):
    text = text.split()
    unknown_res = []
    for word in text:
        unknown_res.append(spell.unknown([word]) != set())
    return unknown_res.count(True)

full_dataset3['count_misspell'] =
                        full_dataset3['text_preproc'].apply(count_misspell)
sns.boxplot(x='hate_speech', y='count_misspell', data=full_dataset3)
```

Notou-se que não há uma predominância de palavras soletradas incorretamente em uma classe ou em outra. Isso provavelmente é devido à forma que comumente se escreve na internet: com abreviações e sem atenção à norma culta da língua. Os resultados foram: 59,46% dos elementos da classe 0 contém palavras escritas incorretamente, enquanto 62,24% dos elementos da classe 1 os contém. O *boxplot* criado é apresentado a seguir:

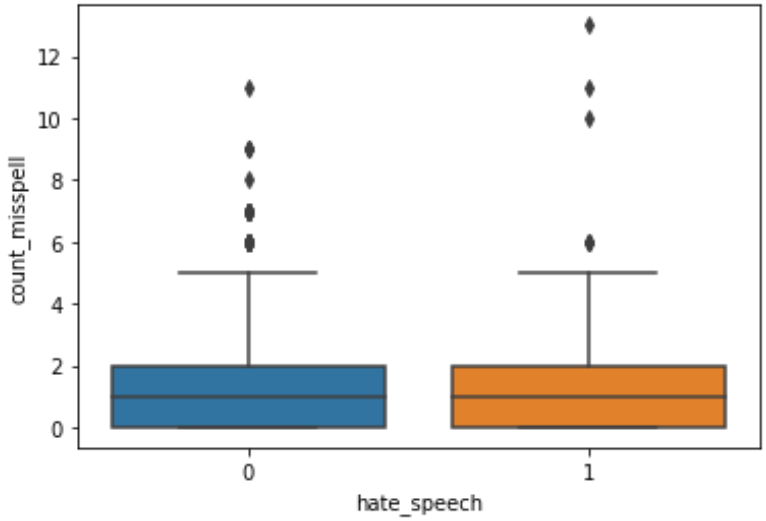

Figura 3: Comparação de contagem de palavras escritas incorretamente entre as classes.

4.3 Presença de palavras de baixo calão

Para contagem de palavras de baixo calão, utilizou-se o método apresentado em 3.2.4: utilizar uma base de dados de palavras ofensivas e fazer a contagem dessas palavras dentro do texto, conforme apresentado a seguir:

```
f = open("datasets\\lista-palavroes-bloqueio.txt", "r", encoding="utf8")
list_swear_words = f.readlines()
list_swear_words = [x.strip() for x in list_swear_words]
f.close()

def count_swear_words(text):
    swear = []
    text = text.split(' ')
    for swear_word in list_swear_words:
        try:
            x = [re.findall('^' + swear_word.lower(), word) for word in
text]
            x = [y for y in x if y != []]
            if x != []:
                swear.append(x[0][0])
        except:
            pass
    return len(swear)

full_dataset3['swear_words'] =
                           full_dataset3['text'].apply(count_swear_words)
```

Com isso, pode-se plotar o *boxplot*:

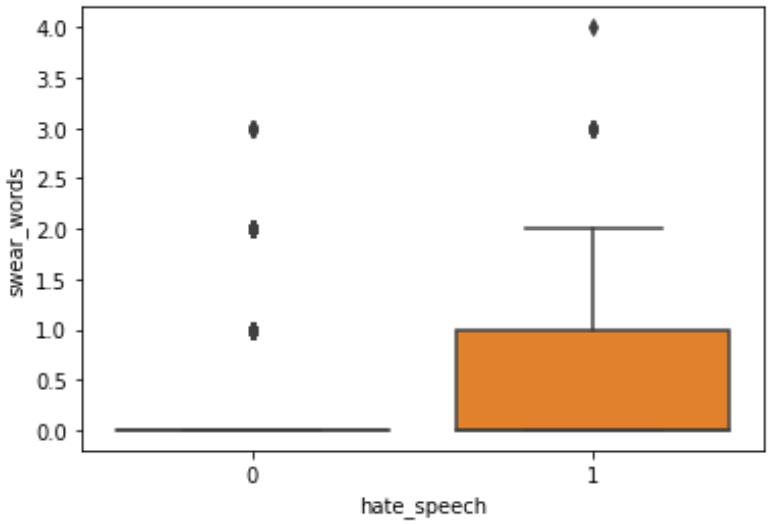

Figura 4: Comparação de quantidade de palavras de baixo calão entre as classes.

Notou-se que há uma prevalência de palavras de baixo calão em discursos o fensivos: em 8,40% dos elementos da classe 0 há palavras ofensivas, enquanto na c lasse 1 isso é verdade para 43,43% dos elementos.

5. Criação de Modelos de *Machine Learning*

Nesta seção serão apresentadas as análises para a criação dos modelos de aprendizado de máquina.

A escolha dos classificadores foi feita com base na sugestão do *website* de tutorial do *scikit-learn*. Com base na Figura 5, os melhores classificadores para predizer uma categoria, utilizando uma base de dados já classificada com menos de 100 mil amostras são LinearSVC e, caso os dados sejam de texto, o Naive Bayes. Além disso, optou-se por analisar também a performance do classificador de árvore de decisão.

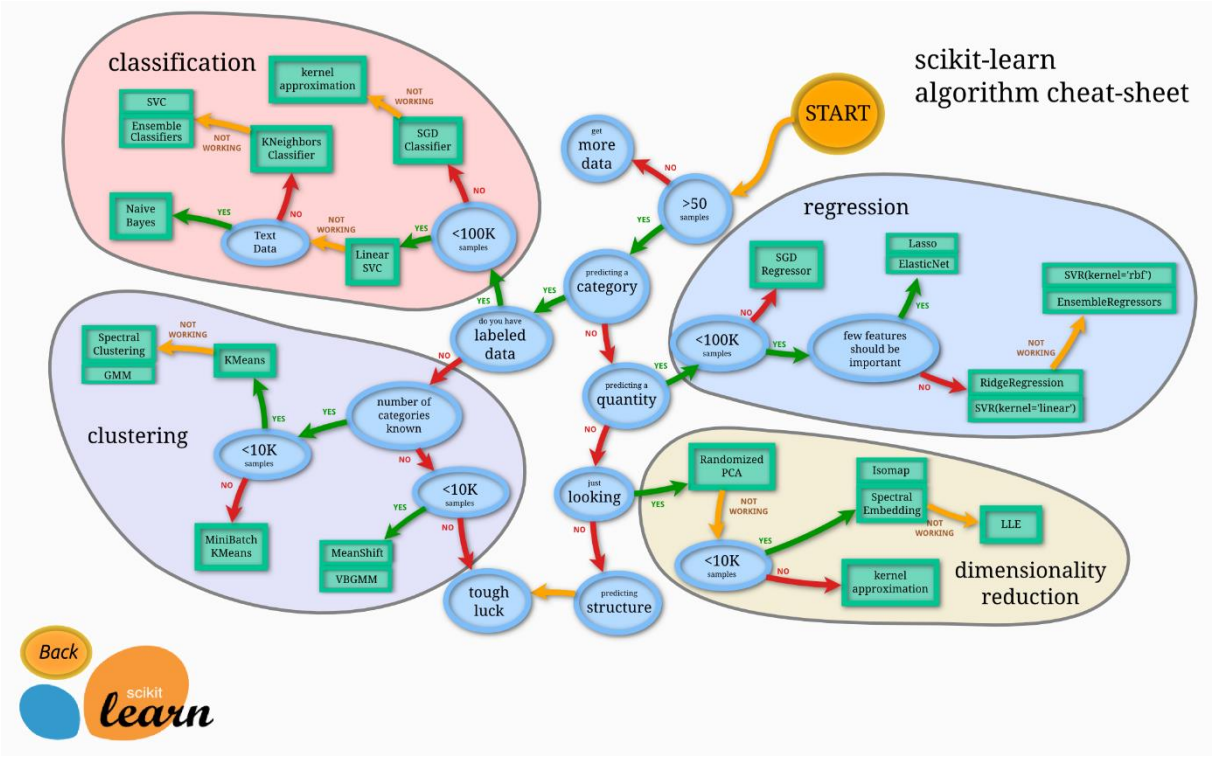

Figura 5: Fluxo de escolha do melhor algoritmo disponível no scikit-learn.
Disponível em https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html.

5.1 Escolhendo os melhores parâmetros para cada classificador

Para escolher os melhores parâmetros de cada classificador, foi realizado uma comparação entre diferentes valores um parâmetro específico para cada classificador, como será visto a seguir. O objetivo era escolher o melhor valor com base no *f1-score* mais alto e no menor *overfitting* entre os valores de treino e teste.

O *f1-score* pode ser considerado como a média ponderada da precisão (razão entre os dados classificados corretamente como positivos e o total de dados classificados como positivos) e do *recall* (razão entre os dados classificados corretamente como positivos e o total de positivos), em que o melhor valor é 1 e o pior é 0. A fórmula do *f1-score* é dada por:

$$f1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (2)$$

Para dividir os dados, foi utilizado o método *train_test_split* de forma a dividir os dados em 60% para dados de treino, 20% para dados de validação e 20% para dados de teste. Mas antes, separou-se os dados de texto e a informação de classificação em diferentes variáveis, para facilitar o trabalho, conforme o trecho de código abaixo. Em seguida, esses dados foram divididos.

```
from sklearn.model_selection import train_test_split
# unbalanced
text3_unb = full_dataset3_unbalanced['text']
class3_unb = full_dataset3_unbalanced['hate_speech']

# balanced translated
text3_tra = full_dataset3_balanced_translate['text']
class3_tra = full_dataset3_balanced_translate['hate_speech']

# balanced repeating
text3_rep = full_dataset3_balanced_repeating['text']
class3_rep = full_dataset3_balanced_repeating['hate_speech']

# balanced undersampling
text3_und = full_dataset3_balanced_undersampling['text']
class3_und = full_dataset3_balanced_undersampling['hate_speech']
```

```
X_train_unb, X_test_unb, y_train_unb, y_test_unb =
                                train_test_split(text3_unb,
                                                 class3_unb,
                                                 test_size=0.2,
                                                 random_state=1)
X_train_unb, X_val_unb, y_train_unb, y_val_unb =
                                train_test_split(X_train_unb,
                                                 y_train_unb,
                                                 test_size=0.25,
                                                 random_state=1)

X_train_tra, X_test_tra, y_train_tra, y_test_tra =
                                 train_test_split(text3_tra,
                                                  class3_tra,
                                                  test_size=0.2,
                                                  random_state=1)
X_train_tra, X_val_tra, y_train_tra, y_val_tra =
                                 train_test_split(X_train_tra,
                                                  y_train_tra,
                                                  test_size=0.25,
                                                  random_state=1)

X_train_rep, X_test_rep, y_train_rep, y_test_rep =
                                 train_test_split(text3_rep,
                                                  class3_rep,
                                                  test_size=0.2,
                                                  random_state=1)
X_train_rep, X_val_rep, y_train_rep, y_val_rep =
                                 train_test_split(X_train_rep,
                                                  y_train_rep,
                                                  test_size=0.25,
                                                  random_state=1)

X_train_und, X_test_und, y_train_und, y_test_und =
                                 train_test_split(text3_und,
                                                  class3_und,
                                                  test_size=0.2,
                                                  random_state=1)
X_train_und, X_val_und, y_train_und, y_val_und =
                                 train_test_split(X_train_und,
                                                  y_train_und,
                                                  test_size=0.25,
                                                  random_state=1)
```

Para a avaliação dos modelos escolhidos, o mesmo processo foi realizado. Foi criada uma *pipeline*, unindo o *CountVectorizor* (utilizando o *preprocessing4*, considerado aqui como o mais completo*), TfIdf* e o classificador em análise. Os parâmetros do classificador variam dentro de um laço de repetição. Por fim, é gerado um gráfico com os valores de *f1-score* em função do parâmetro.

Para comparação dos modelos, o *f1-score* será também utilizado, e foi calculado tanto para os dados de treino quanto para os dados de validação.

5.1.1 LinearSVC

Para avaliar o classificador LinearSVC, escolheu-se o parâmetro de regularização C. A força de regularização é inversamente proporcional a C. Para sua variação, os valores escolhidos foram de 10^{-5} a 10^{3}.

```
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.metrics import f1_score

param_C = np.logspace(-5, 4, 10)

def linear_svc_eval(data):
    train_scores = []
    val_scores = []
    for i in param_C:
        clf = LinearSVC(C=i, max_iter=10000)
        pipeline = Pipeline([
                            ('bow', CountVectorizer(analyzer=prepro-
cessing4, ngram_range=(1,1))),
                            ('tfidf', TfidfTransformer()),
                            ('classifier', clf)
                            ])

        pipeline.fit(data[0], data[2])

        train_predict = pipeline.predict(data[0])
        train_f1 = f1_score(data[2], train_predict, average='weighted')
        train_scores.append(train_f1)

        val_predict = pipeline.predict(data[1])
        val_f1 = f1_score(data[3], val_predict, average='weighted')
        val_scores.append(val_f1)

    return [train_scores, val_scores]
```

Assim, esse resultado foi calculado e plotado para melhor visualização. Com a análise da Figura 6, conclui-se que um bom valor para o parâmetro C, que entrega valores razoáveis de f1-score e com baixo *overfitting* em todas as quatro bases de dados analisadas (*unbalanced, undersampling, repeating* e *translate*), é 10^{-2}. Contudo, mesmo utilizando valores altos de *max_inter* (número máximo de interações), ainda não houve convergência do modelo, informado por meio de um aviso ao rodar o código, conforme apresentado na Figura 7.

```
scores_data3_unbalanced = linear_svc_eval([X_train_unb, X_val_unb,
y_train_unb, y_val_unb])
scores_data3_balanced_translate = linear_svc_eval([X_train_tra, X_val_tra,
y_train_tra, y_val_tra])
scores_data3_balanced_repeating = linear_svc_eval([X_train_rep, X_val_rep,
y_train_rep, y_val_rep])
scores_data3_balanced_undsampl = linear_svc_eval([X_train_und, X_val_und,
y_train_und, y_val_und])
```

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(param_C, scores_data3_unbalanced[0], '-o', label='Train')
axs[0, 0].plot(param_C, scores_data3_unbalanced[1], '-d', label='Test')
axs[0, 0].set_title('Unbalanced data')
axs[0, 1].plot(param_C, scores_data3_balanced_translate[0], '-o', la-
bel='Train')
axs[0, 1].plot(param_C, scores_data3_balanced_translate[1], '-d', label='Test')
axs[0, 1].set_title('Balanced data translation')
axs[1, 0].plot(param_C, scores_data3_balanced_repeating[0], '-o', la-
bel='Train')
axs[1, 0].plot(param_C, scores_data3_balanced_repeating[1], '-d', label='Test')
axs[1, 0].set_title('Balanced data repeating')
axs[1, 1].plot(param_C, scores_data3_balanced_undsampl[0], '-o', label='Train')
axs[1, 1].plot(param_C, scores_data3_balanced_undsampl[1], '-d', label='Test')
axs[1, 1].set_title('Balanced data undersampling')

for ax in axs.flat:
    ax.set(xlabel='param_C', ylabel='F1-score (weighted)')
    ax.set_xscale('log')
plt.subplots_adjust(hspace=0.7, wspace = 0.5)
plt.show()
```

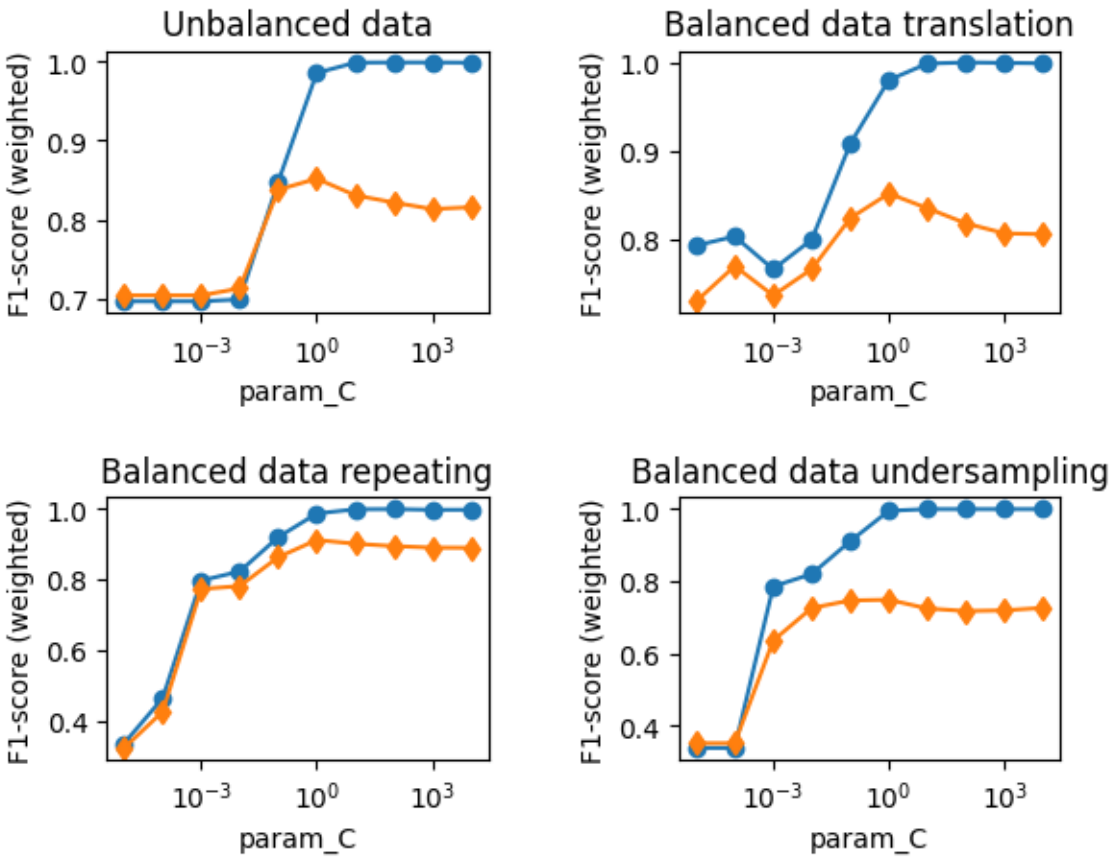

Figura 6: Resultados de f1-score (dados de treino em azul, dados de validação em laranja) com variação do parâmetro C para o classificador *LinearSVC*.

```
c:\users\acer\onedrive\documentos\studies\tcc\hate_speech_detector\venv\lib\site-packages\sklearn\svm\_base.py:985: Convergence
Warning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
c:\users\acer\onedrive\documentos\studies\tcc\hate_speech_detector\venv\lib\site-packages\sklearn\svm\_base.py:985: Convergence
Warning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
c:\users\acer\onedrive\documentos\studies\tcc\hate_speech_detector\venv\lib\site-packages\sklearn\svm\_base.py:985: Convergence
Warning: Liblinear failed to converge, increase the number of iterations.
```

Figura 7: Mensagem de erro de convergência e aumento automático do número de iterações no classificador *LinearSVC*.

Assim, um novo método foi criado para rodar o modelo com cada um dos algoritmos de pré-processamento e com todos os tipos de dados (*unbalanced, undersampling, repeating* e *translate*). Para realizar isso, o método recebe como entrada os dados de treinamento e validação. Para efeitos de comparação, duas *pipelines* foram geradas e avaliadas: uma apenas com o *CountVectorizer* e outra com o *CountVectorizer* e *TfIdfTransformer,* e ambas com o método de *LinearSVC* com parâmetro $C = 10^{-2}$. Essas *pipelines* são ajustadas com base nos dados de treino e, por fim, as

mensagens presentes nos dados de validação são preditas como discurso ofensivo ou não.

Como saída da função, tem-se o valor de *f1-score* da predição dos dados de validação, bem como as duas *pipelines* já ajustadas. Além disso, para analisar se há enviesamento do modelo, optou-se por imprimir na tela a matriz de confusão gerada em cada predição dos dados de validação.

```
lsvc = LinearSVC(C=10**-2)

def linearsvc(X_train, X_val, y_train, y_val, preproc):
    lsvc_bow = Pipeline([
        ('bow', CountVectorizer(analyzer=preproc, ngram_range=(1,1))),
        ('classifier', lsvc),
    ])

    lsvc_bow.fit(X_train, y_train)
    result_val1 = lsvc_bow.predict(X_val)
    print('BOW')
    print(confusion_matrix(y_val, result_val1))
    f1_bow = f1_score(y_val, result_val1, average='weighted')

    lsvc_tfidf = Pipeline([
        ('bow', CountVectorizer(analyzer=preproc, ngram_range=(1,1))),
        ('tfidf', TfidfTransformer()),
        ('classifier', lsvc),
    ])

    lsvc_tfidf.fit(X_train, y_train)
    result_val2 = lsvc_tfidf.predict(X_val)
    print('TF-IDF')
    print(confusion_matrix(y_val, result_val2))
    f1_tfidf = f1_score(y_val, result_val2, average='weighted')

    return f1_bow, f1_tfidf, lsvc_tfidf, lsvc_tfidf
```

5.1.2 Multinomial Naive Bayes

Na avaliação do modelo *Multinomial Naive Bayes,* escolheu-se o parâmetro de suavização aditivo alpha. Os valores escolhidos foram de 1 a 10. E assim foi feito, de forma muito semelhante ao caso anterior:

```
alpha_values = range(1, 11)
def multinomial_nb_eval(data):
    train_scores = []
    val_scores = []
    for i in alpha_values:
        clf = MultinomialNB(alpha=i)
        pipeline = Pipeline([
                            ('bow', CountVectorizer(analyzer=prepro-
cessing4, ngram_range=(1,1))),  # strings to token integer counts
                            ('tfidf', TfidfTransformer()),
                            ('classifier', clf)
                            ])

        pipeline.fit(data[0], data[2])

        train_predict = pipeline.predict(data[0])
        train_f1 = f1_score(data[2], train_predict, average='weighted')
        train_scores.append(train_f1)

        val_predict = pipeline.predict(data[1])
        val_f1 = f1_score(data[3], val_predict, average='weighted')
        val_scores.append(val_f1)

    return [train_scores, val_scores]
```

Assim, esse resultado foi calculado e plotado para melhor visualização. Com a análise da Figura 8, conclui-se que um bom valor para o parâmetro alpha, que entrega valores razoáveis de *f1-score*, é 2.

```
scores_data3_unbalanced = multinomial_nb_eval([X_train_unb, X_val_unb,
y_train_unb, y_val_unb])
scores_data3_balanced_translate = multinomial_nb_eval([X_train_tra,
X_val_tra, y_train_tra, y_val_tra])
scores_data3_balanced_repeating = multinomial_nb_eval([X_train_rep,
X_val_rep, y_train_rep, y_val_rep])
scores_data3_balanced_undersampl = multinomial_nb_eval([X_train_und,
X_val_und, y_train_und, y_val_und])
```

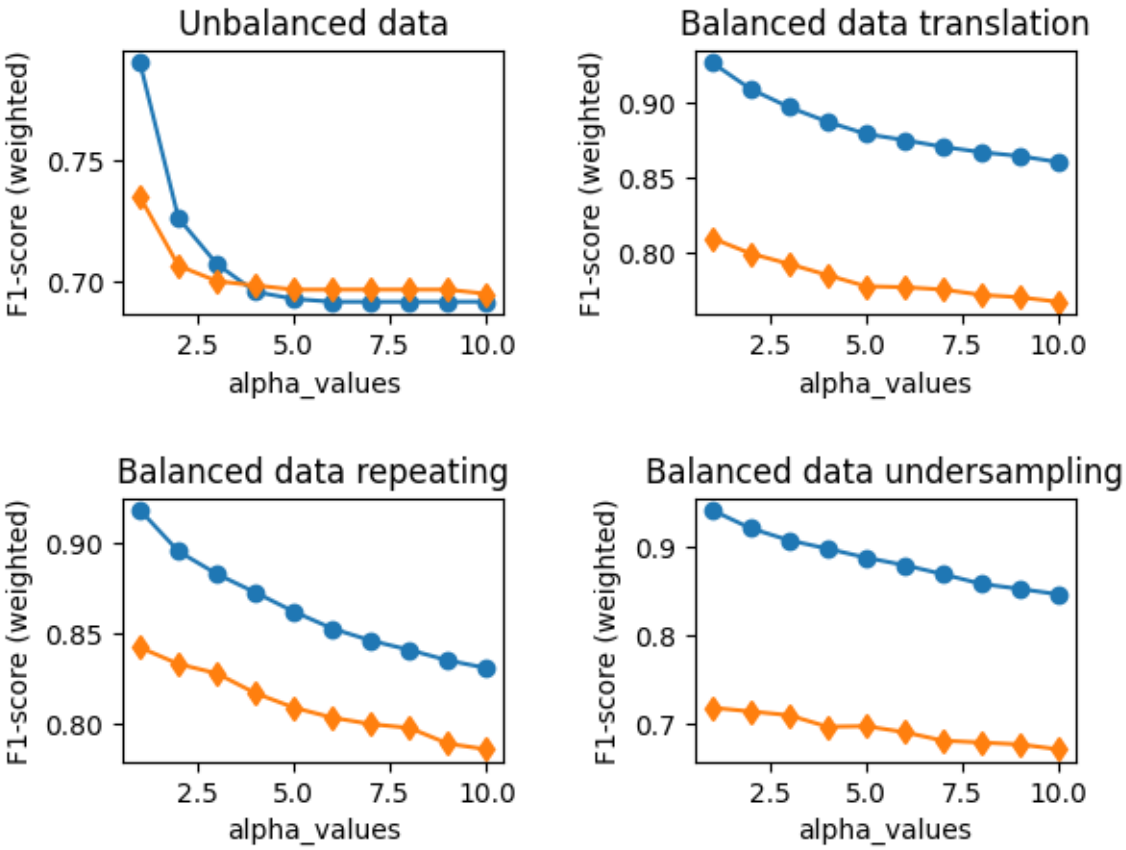

Figura 8: Resultados de f1-score (dados de treino em azul, dados de validação em laranja) com variação do parâmetro alpha para o classificador MultinomialNB.

Com o parâmetro *alpha* escolhido, criou-se o método para rodar todos os algoritmos de pré-processamento em todos os tipos de dados. Assim como o caso anterior, a entrada são os dados de treinamento e validação, e a saída da função é o valor de *f1-score* da predição dos dados de validação, bem como as duas *pipelines* já ajustadas. As *pipelines* criadas utilizam o método *Naive Bayes* com o parâmetro *alpha* igual a 2, sendo que uma contém apenas *CountVectorizer* e outra tem o *CountVectorizer* e *TfIdfTransformer.*

```
mnb = MultinomialNB(alpha=2)

def multinomial_nb(X_train, X_val, y_train, y_val, preproc):
    mnb_bow = Pipeline([
        ('bow', CountVectorizer(analyzer=preproc, ngram_range=(1,1))),
        ('classifier', mnb),
    ])

    mnb_bow.fit(X_train, y_train)
    result_val1 = mnb_bow.predict(X_val)
    print('BOW')
    print(confusion_matrix(y_val, result_val1))
    f1_bow = f1_score(y_val, result_val1, average='weighted')

    mnb_tfidf = Pipeline([
        ('bow', CountVectorizer(analyzer=preproc, ngram_range=(1,1))),
        ('tfidf', TfidfTransformer()),
        ('classifier', mnb),
    ])

    mnb_tfidf.fit(X_train, y_train)
    result_val2 = mnb_tfidf.predict(X_val)
    print('TF-IDF')
    print(confusion_matrix(y_val, result_val2))
    f1_tfidf = f1_score(y_val, result_val2, average='weighted')

    return f1_bow, f1_tfidf, mnb_bow, mnb_tfidf
```

5.1.3 Árvore de decisão

Por último, para a o classificador em árvore de decisão, optou-se por variar o parâmetro *max_depth*, que indica a máxima profundidade da árvore. A variação foi de 1 a 10. De forma semelhante aos casos anteriores, foi plotado o valor de f1-score para os dados de treino e de validação em função dos valores de *max_depth*.

Observa-se que não houve variação no f1-score no caso de *unbalanced data*, mas para os outros dados há um aumento nesse valor. O modelo de árvore de decisão é conhecido por gerar *overfitting* quando o parâmetro de máxima profundidade é alto, pois o modelo fica mais específico para os dados de treinamento. Para evitar o

overfitting, mas ainda apresentar bons valores de *f1-score*, o valor de *max_depth* escolhido foi de 5.

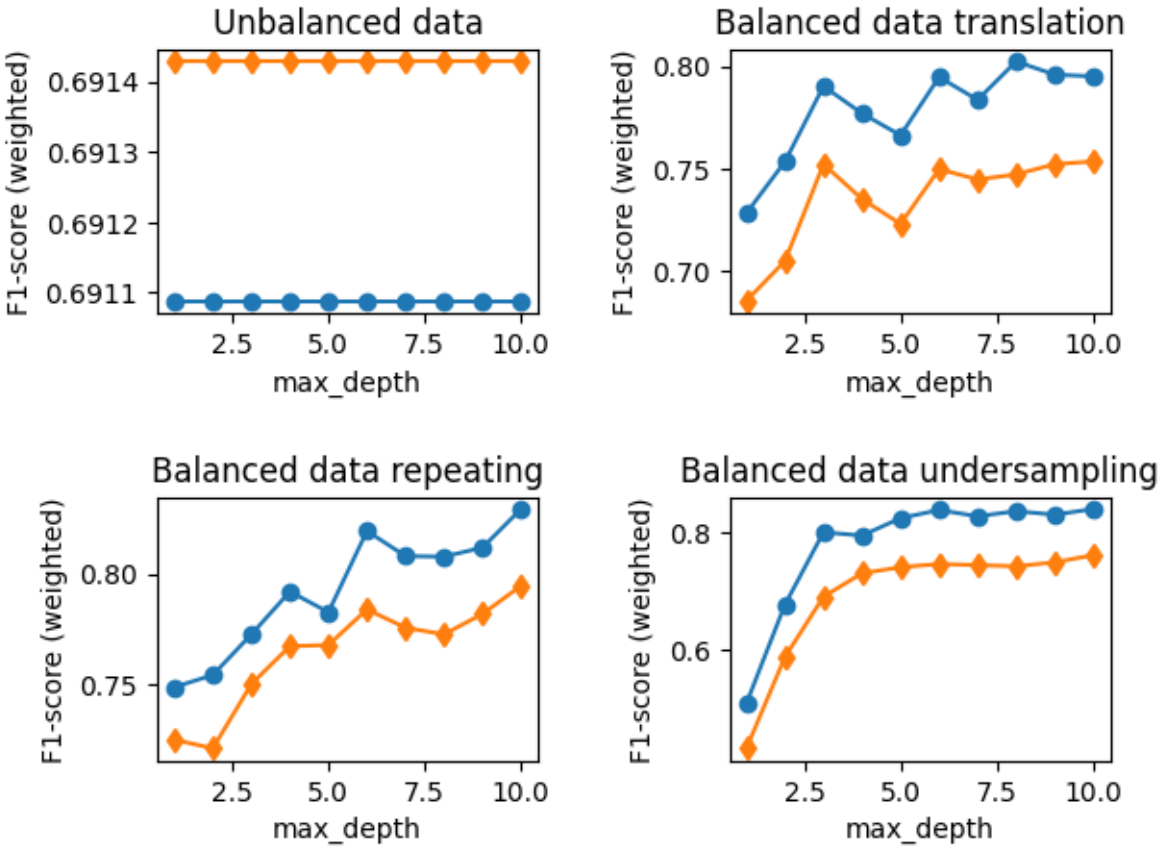

Figura 9: Resultados de f1-score (dados de treino em azul, dados de validação em laranja) com variação do parâmetro *max_depth* para o classificador *RandomForestClassifier*.

Da mesma forma que os casos anteriores, um método foi criado de forma a rodar duas *pipelines* para cada conjunto de tipo de pré-processamento e base de dados, conforme abaixo:

```
rfc = RandomForestClassifier(max_depth=5)

def random_forest(X_train, X_val, y_train, y_val, preproc):
    rfc_bow = Pipeline([
        ('bow', CountVectorizer(analyzer=preproc, ngram_range=(1,1))),
        ('classifier', rfc),
    ])

    rfc_bow.fit(X_train, y_train)
    result_val1 = rfc_bow.predict(X_val)
    print('BOW')
    print(confusion_matrix(y_val, result_val1))
    f1_bow = f1_score(y_val, result_val1, average='weighted')

    rfc_tfidf = Pipeline([
        ('bow', CountVectorizer(analyzer=preproc, ngram_range=(1,1))),
        ('tfidf', TfidfTransformer()),
        ('classifier', rfc),
    ])

    rfc_tfidf.fit(X_train, y_train)
    result_val2 = rfc_tfidf.predict(X_val)
    print('TF-IDF')
    print(confusion_matrix(y_val, result_val2))
    f1_tfidf = f1_score(y_val, result_val2, average='weighted')

    return f1_bow, f1_tfidf, rfc_bow, rfc_tfidf
```

5.2 Avaliando os classificadores

Com isso, todas as *pipelines* puderam ser rodadas. Os valores de *f1-score* obtidos em cada um dos casos analisados são apresentados abaixo, onde RF é o modelo *RandomForest*, MNB é o *Multinomial Naive Bayes* e LSVC é o *Linear SVC*.

Os resultados de *f1-score* para os dados de validação são apresentados na Tabela 6.

Apesar dos dados desbalanceados terem apresentado um bom resultado, os números são interessantes: o *f1-score* utilizando o modelo *RandomForest* foi o mesmo para todos os tipos de pré-processamento. Isso ocorreu pois os dados da classe majoritária foram todos classificados corretamente, enquanto os da minoritária foram todos classificados erroneamente – isso foi visto na matriz de confusão dos classificadores com essa base de dados. Em outras palavras, as mensagens que não foram consideradas ofensivas (classe majoritária) foram quase sempre classificadas corretamente, enquanto as mensagens ofensivas foram classificadas como não-ofensivas praticamente todas as vezes. Esse comportamento já era esperado, pois base de

dados desbalanceadas podem enviesar o modelo para a classe majoritária, conforme explanado anteriormente.

Tabela 6: Resultados de f1-score para os dados de validação.

f1-scores		unbalanced		undersampling		repeating		translate	
		CountVec	TF-IDF	CountVec	TF-IDF	CountVec	TF-IDF	CountVec	TF-IDF
preproces-sing1	RF	0,6914	0,6914	0,7294	0,7097	0,7574	0,7521	0,7309	0,7016
	MNB	0,8213	0,7100	0,7316	0,7279	0,8491	0,8291	0,8111	0,8069
	LSVC	0,8284	0,6967	0,7511	0,7706	0,8462	0,7695	0,8009	0,7607
preproces-sing2	RF	0,6914	0,6914	0,7558	0,7116	0,7775	0,7712	0,7196	0,7250
	MNB	0,8184	0,7066	0,7302	0,7146	0,8475	0,8357	0,8112	0,7972
	LSVC	0,8269	0,6967	0,7799	0,7745	0,8371	0,7989	0,7985	0,7647
preproces-sing3	RF	0,6914	0,6914	0,7487	0,7450	0,7499	0,7693	0,7307	0,7237
	MNB	0,7920	0,7002	0,7382	0,7305	0,8435	0,8200	0,8055	0,7985
	LSVC	0,8109	0,7194	0,7560	0,7530	0,8351	0,7646	0,7919	0,7378
preproces-sing4	RF	0,6914	0,6914	0,7522	0,7468	0,7744	0,7782	0,7229	0,7405
	MNB	0,8162	0,7066	0,7337	0,7142	0,8456	0,8327	0,8121	0,7991
	LSVC	0,8280	0,6985	0,7821	0,7658	0,8381	0,7935	0,7961	0,7657

Para a avaliação dos dados de teste, criou-se o método *predict_test*, que recebe os dados a serem preditos e a pipeline a ser utilizada para classificação. O retorno é o *f1-score*.

```
def predict_test(X, y, pipeline):
    result = pipeline.predict(X)
    print(confusion_matrix(y, result))
    return f1_score(y, result, average='weighted')
```

Assim, foi possível criar mais uma tabela com os resultados de *f1-score* para os dados de teste, com os melhores resultados de cada base de dados marcados em negrito, como nota-se na Tabela 7.

De posse desses resultados, nota-se que os valores de *f1-score* do classificador *RandomForest* com os dados *unbalanced* são iguais, e isso ocorre pelo mesmo motivo dos dados de validação: as mensagens da classe majoritária foram todas classificadas corretamente, enquanto os da classe minoritária foram classificados erroneamente. Os outros modelos criados com base nos dados desbalanceados são, no geral, piores em relação aos outros. Os modelos criados com os dados *undersampling* também resultaram em um *f1-score* relativamente baixo. De fato, uma redução na quantidade de dados de treinamento pode gerar problemas de generalização, visto que o modelo foi apresentado a poucos dados.

Os modelos criados com base nos dados *repeating* apresentam os melhores valores de *f1-score*. Contudo, como os dados foram repetidos, há alta probabilidade de enviesamento, o que de fato pode aumentar os *scores*. Já o modelo com base em *translate*, apesar de apresentar *f1-score* menor, pode ser mais generalista, visto que foi treinado com dados não repetidos, pois a classe minoritária foi aumentada com adição de alguns elementos aleatórios.

Tabela 7: Resultados de f1-score para os dados de teste.

f1-scores		unbalanced		undersampling		repeating		translate	
		CountVec	TF-IDF	CountVec	TF-IDF	CountVec	TF-IDF	CountVec	TF-IDF
preprocessing1	RF	0,7019	0,7019	0,7154	0,7337	0,7349	0,7411	0,7206	0,7136
	MNB	**0,7904**	0,7191	0,6908	0,7169	0,7713	0,7972	0,7964	**0,8113**
	LSVC	0,7036	0,7036	0,7566	0,7566	0,7671	0,7671	0,7635	0,7635
preprocessing2	RF	0,7019	0,7019	0,7346	0,7325	0,7602	0,7658	0,7176	0,7177
	MNB	0,7878	0,7158	0,7193	0,7272	0,7809	**0,8067**	0,7863	0,8064
	LSVC	0,7106	0,7106	0,7674	0,7674	0,7948	0,7948	0,7633	0,7633
preprocessing3	RF	0,7019	0,7019	0,7066	0,7480	0,7531	0,7500	0,7092	0,7184
	MNB	0,7572	0,7019	0,6948	0,7173	0,7578	0,7900	0,7833	0,8023
	LSVC	0,7268	0,7268	**0,7677**	**0,7677**	0,7555	0,7555	0,7351	0,7351
preprocessing4	RF	0,7019	0,7019	0,7461	0,7645	0,7635	0,7690	0,7262	0,7436
	MNB	0,7865	0,7158	0,7212	0,7328	0,7778	0,8053	0,7832	0,8039
	LSVC	0,7106	0,7106	0,7655	0,7655	0,7903	0,7903	0,7590	0,7590

Portanto, decidiu-se utilizar o modelo treinado com os dados *translate* com melhor performance, isto é, o *Multinomial Naive Bayes*, com o pré-processamento *preprocessing1* (apenas com limpeza de dados e remoção de *stopwords*) e TF-IDF.

É interessante notar que os pré-processamentos mais complexos (com stemização ou destaque de palavras chaves) não aumentaram a performance de forma significativa, de forma que o melhor modelo foi aquele com pré-processamento mais simples.

Assim, o modelo escolhido treinado foi salvo no arquivo *mnb_tfidf_*tra1, utilizando o Joblib, conforme apresentado em (MACHINE LEARNING MASTERY, 2021) e mostrado abaixo:

```
# saving the best model: mnb_tfidf_tra1
import joblib
filename = 'mnb_tfidf_tra1.sav'
joblib.dump(mod, filename)
```

6. Apresentação dos Resultados

Conforme visto na sessão anterior, obteve-se bons resultados nos dados de teste e de validação. Porém, para melhor avaliação do modelo, o próximo passo foi usá-lo para classificar novas mensagens escritas na internet. Para isso, coletou-se dados da rede social *Twitter*, na madrugada entre os dias 25 e 26 de abril de 2021. Os temas mais comentados foram relacionados à premiação do Oscar, a dois *reality shows*: "*Big Brother Brasil*" e "No Limite", além dos jogos de futebol que ocorreram no domingo.

Para coleta das mensagens, utilizou-se a biblioteca *Tweepy*, para Python. A ideia foi mapear os 20 temas mais comentados no Brasil e coletar 1000 *tweets* em português para cada tema. Isso foi feito da seguinte forma:

```
import tweepy
import tqdm

BRAZIL_WOE_ID = 23424768
brazil_trends = api.trends_place(BRAZIL_WOE_ID)[0]['trends'][:20]

tweets_df = pd.DataFrame(columns=['topic', 'text', 'RT', 'time'])

qt_items = 1000

for brazil_trend in brazil_trends:
    search_words = brazil_trend['name']
    print(search_words)

    tweets = tweepy.Cursor(api.search,
                           q=search_words,
                           tweet_mode = "extended",
                           lang="pt").items(qt_items)

    for tweet in tqdm.tqdm(tweets):
        try:
            text = tweet.retweeted_status.full_text
            rt = "1"
        except AttributeError:
            text = tweet.full_text
            rt = "0"

        tweets_df = tweets_df.append({'topic':search_words,
                                      'text': text,
                                      'RT': rt,
                                      'time': tweet.created_at},
                                     ignore_index=True)
```

Dessa forma, cria-se um *dataframe* com 20.000 entradas, com as colunas: *topic* (nome do tópico), *text* (mensagem/*tweet*), *RT* (informação se a mensagem surgiu oriunda de um *retweet*, ou seja, se ela é original ou não) e *time* (data e hora da publicação).

Em seguida, importa-se o modelo treinado, faz-se a predição de classificação de discurso de ódio (com o método *predict*), adiciona-se o resultado a uma nova coluna do *dataframe* e, por fim, se salva o *dataframe* em um arquivo do tipo csv, conforme apresentado abaixo:

```
loaded_model = joblib.load("mnb_tfidf_tra1.sav")
tweets_df['hate'] = loaded_model.predict(tweets_df['text'])
tweets_df.to_csv('tweets.csv', index=False)
```

Com isso, foi possível usar a ferramenta Power BI para visualização dos dados.

Notou-se que praticamente 70% dos *tweets* coletados foram oriundos de *retweets* enquanto 30% eram mensagens originais. Além disso, 35% das mensagens foram classificadas como discurso ofensivo. Essas informações podem ser observadas nas Figuras 10 e 11, respectivamente.

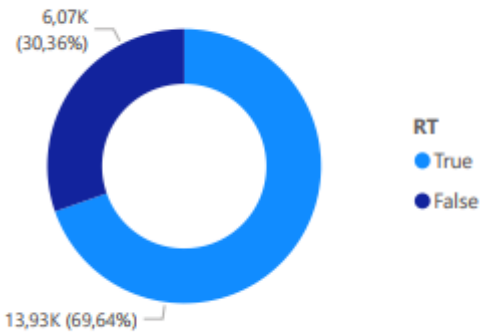

Figura 10: Relação de *retweets* e mensagens originais.

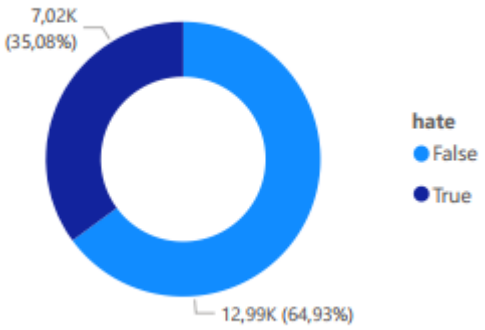

Figura 11: Classificação das mensagens como discurso ofensivo ou não-ofensivo.

Outra análise feita foi verificar o discurso ofensivo em cada um dos tópicos mais comentados no país. Os três tópicos com mais mensagens classificadas negativamente, conforme observado na Figura 12, foram as seguintes: *zendaya* (em referência à atriz que apresentava a premiação do Oscar naquela noite), *André* (participante anunciado em um novo *reality show*) e *#ForaArthur* (participante de outro *reality* que estava sendo cotado para ser eliminado do programa).

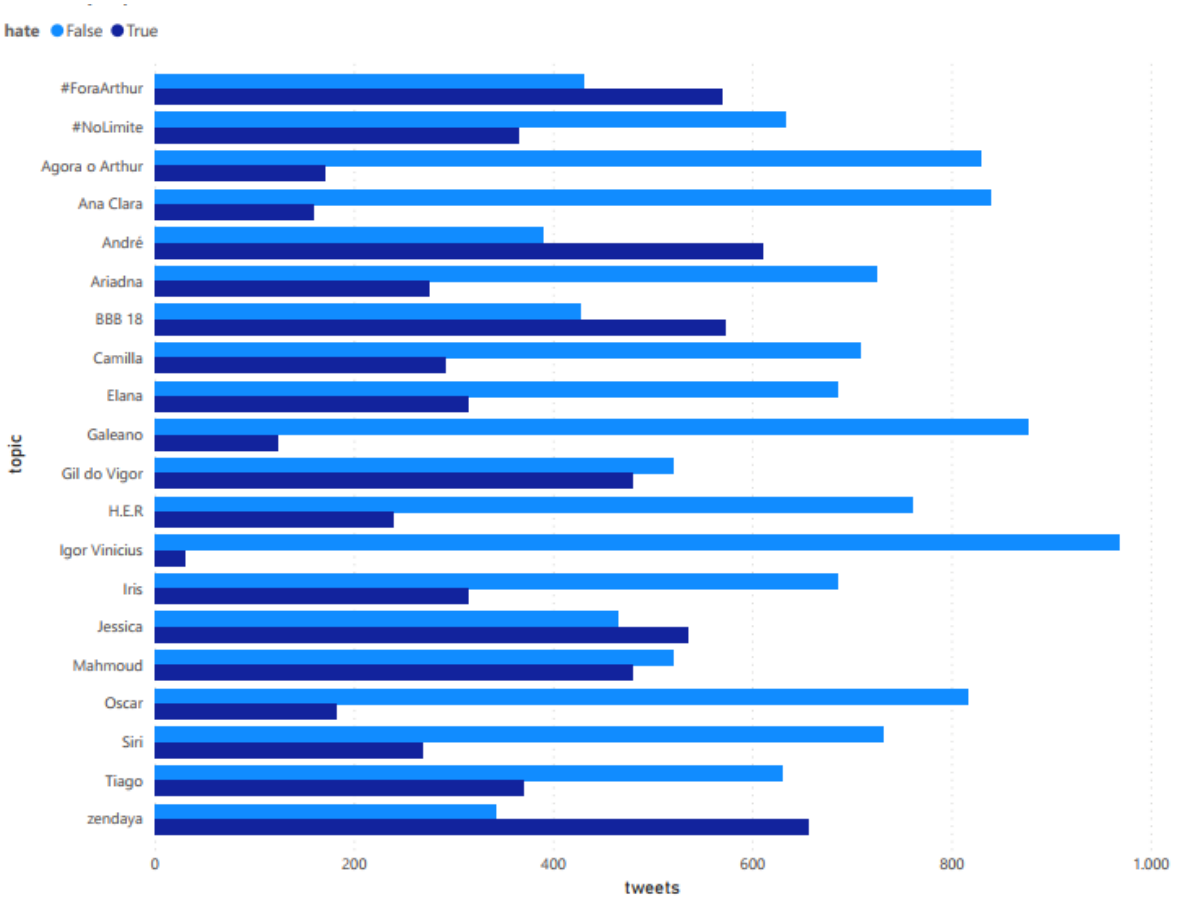

Figura 12: Classificação de discurso ofensivo por tópico.

7. Considerações Finais

Por fim, os novos dados e classificações foram manualmente checados para melhor análise final do modelo proposto. Como amostra do resultado, tem-se o seguinte:

Mensagem classificada corretamente como não ofensiva:
“Ariadna foi extremante injustiçada, sendo a primeira eliminada do BBB 11 após uma série de ações transfóbicas. Em seu Instagram, ela deixou um desabafo sobre essa nova oportunidade no #NoLimite antes de ser confinada. SIGAM A QUEEN!”

Mensagem classificada corretamente como ofensiva:
“E old que o top 3 do ""no limite"" é a Angelica, Elana e Gleice. Ver essa temporada só pra ver o viado xexelento do mahmoud sofrendo”

Mensagem classificada erroneamente como não ofensiva (falso negativo):
*“MULHER???? VAI GANHAR A P*** DE UMA PROVA SUA CRETINA AZEDA”*

Mensagem classificada erroneamente como ofensiva (falso positivo):
Zendaya é uma das atrizes mais belas e talentosas de sua geração #Oscars

Notou-se que falsos positivos foram comuns, especialmente quando havia muitos elogios (exatamente o oposto de um discurso ofensivo) no comentário. Isso provavelmente se deve ao fato que os dados de treino foram, em sua maioria, coletados a partir de *hashtags* e temas que indicariam uma maior quantidade de discurso ofensivo. Em outras palavras, o modelo conseguiu distinguir bem o que era ofensivo e teve poucos falsos negativos (identificados empiricamente). Contudo, por não ter tido contato com discursos elogiosos, o modelo passou a classificá-los como ofensivos, gerando um alto número de falsos positivos. Uma possível solução seria retreinar o modelo com mais discursos favoráveis – como, por exemplo, as mensagens direcionadas à atriz Zendaya.

A subjetividade do assunto “discurso ofensivo/discurso de ódio” de fato torna o seu estudo mais complexo, em especial quando o objetivo é criar modelos de aprendizado de máquina os que identificam automaticamente. Além disso, há na internet uma predominância de discursos informais, com abreviações, gírias e escritas sem atenção a uma norma linguística. Contudo, a importância ética deste problema social faz com que esse tema precise ser mais bem discutido, estudado e explorado não apenas pelas empresas que podem sofrer impactos financeiros, mas também pela comunidade cientifica de diversas áreas e até mesmo pela população geral.

8. Links

Link para o vídeo:

https://www.youtube.com/watch?v=9pq6f2LfjuM

Link para o repositório:

https://github.com/luciananobrega/hate_speech_detector

REFERÊNCIAS

AROYEHUN, S. T.; GELBUKN, A. **Aggression Detection in Social Media: Using Deep Neural Networks, Data Augmentation, and Pseudo Labeling.** First Workshop on Trolling, Aggression and Cyberbullying (TRAC-2018), v. 1, p. 90-97, 2018.

BRADESCO. **Novas respostas da Bia contra o assédio.** Disponível em: < https://banco.bradesco/aliadosbia/>. Acesso em 17 de abril de 2021.

EDUCATIVE. **CountVectorizer in Python.** Disponível em: <https://www.educative.io/edpresso/countvectorizer-in-python>. Acesso em 25 de fevereiro de 2021.

FORTUNA, Paula; DA SILVA, João Rocha; SOLER-COMPANY, Juan; WANNER, Leo; NUNES, Sérgio. **Offensive Comments in the Brazilian Web: a dataset and baseline results.** Florença/Itália: Proceedings of the Third Workshop on Abusive Language Online, 2019.

GOOGLE CLOUD. **Cloud Translation Documentation: Quotas and limits**. Disponível em: <https://cloud.google.com/translate/quotas>. Acesso em 28 de fevereiro de 2021.

MACHINE LEARNING MASTERY. **Save and Load Machine Learning Models in Python with scikit-learn**. Disponível em: < https://machinelearningmastery.com/save-load-machine-learning-models-python-scikit-learn/>. Acesso em 17 de abril de 2021.

PEDREGOSA et. Al, **Scikit-learn: Machine Learning in Python.** Journal of Machine Learning Research, Volume 12, páginas 2825-2830, 2011.

PELLE, Rogers P. de; MOREIRA, Viviane P.. **Offensive Comments in the Brazilian Web: a dataset and baseline results.** Porto Alegre: Sociedade Brasileira de Computação, 2017.

RANZI, Carlos Eduardo. **lista-palavroes-bloqueio.txt**. Disponível em: <https://pt.scribd.com/document/345921799/lista-palavroes-bloqueio-txt>. Acesso em 2 de agosto de 2020.

REDAÇÃO GQ, GLOBO. **Stop Hate for Profit: ativistas pressionam empresas brasileiras a aderirem ao boicote contra o Facebook**. Disponível em: <https://gq.globo.com/Prazeres/Poder/noticia/2020/07/stop-hate-profit-ativistas-pressionam-empresas-brasileiras-aderirem-ao-boicote-contra-o-facebook.html>. Acesso em 8 de setembro de 2020.

SCIKIT-LEARN, **Choosing the right estimator.** Disponível em <https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html>. Acesso em 5 de fevereiro de 2021.

TOWARDS DATA SCIENCE. **Accuracy, precision or f1?.** Disponível em: <https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9>. Acesso em 5 de fevereiro de 2021.