Report

Journal Paper Used:

Headline : A systematic review of hate speech automatic detection using natural language processing

Link : <https://www.sciencedirect.com/science/article/pii/S0925231223003557?ref=pdf_download&fr=RR-2&rr=88226281ce718a1e>

Dataset Used:

# Headline : **Hate Speech Detection curated Dataset🤬**

About the Data : Social media platforms have become the most prominent medium for spreading hate speech, primarily through hateful textual content. An extensive dataset containing emoticons, emojis, hashtags, slang, and contractions is required to detect hate speech on social media based on current trends. This dataset contains hate speech sentences in English and is confined into two classes, one representing hateful content and the other representing non-hateful content.

|  |  |
| --- | --- |
|  | **Specifications table** |
| Subject | Natural Language Processing - NLP |
| Specific subject area | A curated dataset comprising emojis, emoticons, and contractions bundled into two classes, hateful and non-hateful, to detect hate speech in text. |
| Type of data | Text |
| Data format | Annotated, Analysed, Filtered Data |
| Data Article | [A curated dataset for hate speech detection on social media text](https://www.sciencedirect.com/science/article/pii/S2352340922010356) |
| Data source location | <https://data.mendeley.com/datasets/9sxpkmm8xn/1> |

-**Value of this Data :**

1. This dataset is useful for training machine learning models to identify hate speech on social media in text. It reflects current social media trends and the modern ways of writing hateful text, using emojis, emoticons, or slang. It will help social media managers, administrators, or companies develop automatic systems to filter out hateful content on social media by identifying a text and categorizing it as hateful or non-hateful speech.
2. Deep Learning (DL) and Natural Language Processing (NLP) practitioners can be the target beneficiaries as this dataset can be used for detecting hateful speech through DL and NLP techniques. Here the samples are composed of text sentences and labels belonging to two categories “0″ for non-hateful and “1″ for hateful.
3. Additionally, this data set can be used as a benchmark data set to detect hate speech
4. The data set is neutralized in such a way that it can be used by anyone as it doesn't include any entities or names which can have an impact or cyber harm on the user that generated the content. Researchers can take advantage of the pre-processed dataset for their projects as it maintains and follows the policy guidelines.

Link:

https://www.kaggle.com/datasets/waalbannyantudre/hate-speech-detection-curated-dataset/data

Research Gaps:

1. Multilingual Hate Speech Detection:

Further research is needed to evaluate the performance of deep learning models on multilingual hate speech detection. This could involve collecting and annotating datasets from different languages and evaluating the performance of deep learning models on these datasets.

1. Hate Speech Detection in Different Domains and Contexts:

Further research is needed to evaluate the performance of deep learning models on hate speech detection in different domains and contexts. This could involve collecting and annotating datasets from different domains and contexts and evaluating the performance of deep learning models on these datasets.

1. Real-World Hate Speech Detection:

Further research is needed to evaluate the performance of deep learning models on hate speech detection in real-world scenarios, such as online platforms and social media networks. This could involve deploying deep learning models on these platforms and evaluating their performance in detecting hate speech in real-time.

Explanation and Way to fill the gaps:

For the given Research paper, the research gaps could be filled in several ways. One way is to address the need for more multilingual datasets, which are not widely available. These datasets can be collected and annotated by researchers and data enthusiasts.

Another approach could be to evaluate the performance of different deep learning models, such as BERT and RoBERTa, on multilingual hate speech detection datasets. This can be done by comparing their performance metrics, such as accuracy, precision, recall, and F1-score.

Additionally, further research could be conducted on evaluating the performance of these models in different domains and contexts, such as online platforms, social media networks, and news websites. This could involve evaluating their performance on hate speech detection in real-time scenarios and understanding the factors that contribute to their effectiveness.

Another research gap that could be addressed is the need for more diverse datasets. This can be achieved by collecting and annotating datasets from diverse linguistic and cultural backgrounds. These datasets can then be used to train deep learning models for hate speech detection, leading to more accurate and reliable models.

Furthermore, the effectiveness of pre-trained models and their ability to adapt to new languages could be investigated. This can be done by fine-tuning pre-trained models on new datasets and evaluating their performance.

Finally, the research gaps in the given paper could be addressed by investigating the limitations of current deep learning models in detecting hate speech. This can involve studying the types of hate speech that these models struggle to detect and understanding the reasons behind their inadequacy. This can lead to the development of more effective models and techniques for detecting hate speech.

These approaches can be complemented by exploring other aspects of hate speech detection, such as detecting the presence of toxicity and other harmful behaviors in online discussions. This can be done by training deep learning models on datasets containing examples of toxicity and other harmful behaviors. These models can then be used to detect hate speech, toxicity, and other harmful behaviors in online discussions.

This approach involves conducting further research in the area of hate speech detection, particularly in multilingual settings, and exploring other aspects of hate speech detection that have not been thoroughly investigated in the literature. It aims to address the research gaps identified in the given Research paper and contribute to the advancement of the field of hate speech detection.

Code :

In this notebook, I aim to explore various methods for detecting hateful speech within the dataset.

1. TF-IDF with Linear Regression model
2. Bert model with native Pytorch
3. Fine-tuning a model with the Trainer API

IMPORT LIBRARIES:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import os

import re

import string

from pathlib import Path

from collections import Counter

import random

import operator

from tqdm import tqdm

import time

from wordcloud import WordCloud

from string import punctuation

import nltk

import subprocess

from nltk.corpus import stopwords

from nltk.corpus import wordnet

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer, SnowballStemmer

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix, precision\_score, recall\_score, f1\_score

import torch

from torch import nn, optim

from torch.nn import functional as F

from torch.utils.data import Dataset, DataLoader, RandomSampler, SequentialSampler

from transformers import BertModel, BertTokenizer

from transformers import AutoModelForSequenceClassification

from transformers import TrainingArguments, Trainer

from transformers import DataCollatorWithPadding

import evaluate

%matplotlib inline

# **Load the Data**

data\_full = pd.read\_csv('HateSpeechDatasetBalanced.csv')

num\_rows, num\_cols = data\_full.shape

print(f'Rows: {num\_rows},  columns: {num\_cols}')

data\_full.sample(3)

Rows: 726119, columns: 2

[3]:

|  | **Content** | **Label** |
| --- | --- | --- |
| **386432** | you have no authority to be threatening to blo... | 0 |
| **253721** | demographics of lebanon thanks for experimenti... | 0 |
| **96406** | fuck it i hate my husband | 0 |

data\_full.groupby('Label').count()

|  |  |
| --- | --- |
|  | **Content** |
| **Label** |  |
| 0 | 361594 |
| 1 | 364525 |

#The dataset is quite big so we take just 4000 examples of each class

df1 = data\_full.query('Label == 0').sample(4000)

df2 = data\_full.query('Label == 1').sample(4000)

data = pd.concat([df1, df2], ignore\_index=True)

data.shape

(8000, 2)

# **Data Exploration**

Let's take a look on the most frequent words for each class

combined\_title = ' '.join(df1['Content'])

wordcloud\_img = WordCloud(width = 800, height = 800,

                            background\_color ='Black', colormap = 'BuGn',

                            min\_font\_size = 10).generate(combined\_title)

plt.figure(figsize=(10,10))

plt.imshow(wordcloud\_img)

plt.axis('off')

plt.title('Frequent words in Non-Hateful Comments')

plt.tight\_layout(pad=2)

plt.show()



combined\_title = ' '.join(df2['Content'])

wordcloud\_img = WordCloud(width = 800, height = 800,

                            background\_color ='Yellow', colormap = 'hot\_r',

                            min\_font\_size = 10).generate(combined\_title)

plt.figure(figsize=(10,10))

plt.imshow(wordcloud\_img)

plt.axis('off')

plt.title('Frequent words in Hateful Comments')

plt.tight\_layout(pad=2)

plt.show()



# **TF-IDF with Linear Regression model**

* This approach can be considered as baseline.
* Firstly, we need clean and prepare text: clean, remove stopwords, apply lemmatization OR stemming. By default I choose stemming.
* Then convert text to TF-IDF vector and use it as a feature for simple classification model Logistic Regression.

data\_tfidf = data.copy()

stopwords\_l = stopwords.words('english')

punctuation = re.compile("[" + re.escape(string.punctuation) + "]")

lemmatizer = WordNetLemmatizer()

stemmer = SnowballStemmer('english') #Snowball stemmer initialised

def text\_cleaning(text, mode="stemming"):

    res = []

    text\_clean = re.sub(punctuation,'',text)

    tokens = word\_tokenize(text\_clean)

    for token in tokens:

        if token.lower() not in stopwords\_l:

            if mode == "stemming":

                prepared\_word = stemmer.stem(token)

            else:

                prepared\_word = lemmatizer.lemmatize(token)

            res.append(prepared\_word)

    return ' '.join(res)

data\_tfidf['cleaned\_text'] = data\_tfidf['Content'].apply(text\_cleaning)

train, test = train\_test\_split(data\_tfidf, test\_size=0.3, stratify=data['Label'], random\_state=42)

X\_train = train['cleaned\_text']

y\_train = train['Label']

X\_test = test['cleaned\_text']

y\_test = test['Label']

# TfidfVectorizer

tfidf\_vectorizer = TfidfVectorizer(stop\_words='english')

X\_train\_vect = tfidf\_vectorizer.fit\_transform(X\_train)

X\_test\_vect = tfidf\_vectorizer.transform(X\_test)

linear\_clf = LogisticRegression()

# train the model with training data processed using TF-IDF

linear\_clf.fit(X\_train\_vect, y\_train)

A close-up of a blue and white sign

Description automatically generated

y\_pred\_tf\_idf = linear\_clf.predict(X\_test\_vect)

report = classification\_report(y\_test, y\_pred\_tf\_idf)

print(report)

display(pd.DataFrame({"Predicted: Unhateful": confusion\_matrix(y\_test, y\_pred\_tf\_idf)[:, 0],

              "Predicted: Hateful": confusion\_matrix(y\_test, y\_pred\_tf\_idf)[:, 1]},

             index=['Actual: Unhateful', 'Actual: Hateful']))

precision recall f1-score support

0 0.78 0.76 0.77 1200

1 0.76 0.79 0.78 1200

accuracy 0.77 2400

macro avg 0.77 0.77 0.77 2400

weighted avg 0.77 0.77 0.77 2400

|  |  |  |
| --- | --- | --- |
|  | Predicted: Unhateful | Predicted: Hateful |
| Actual: Unhateful | 908 | 292 |
| Actual: Hateful | 253 | 947 |

# **Bert model with native Pytorch**

Now I want to train Bert model with classification head with native Pytorch.

train, validation = train\_test\_split(data, test\_size=0.3, stratify=data['Label'], random\_state=42)

# Define Dataset

class HateSpeechDataset(Dataset):

    def \_\_init\_\_(self, data):

        # Initialize BERT tokenizer

        self.tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

        self.data = data

    def \_\_len\_\_(self):

        return len(self.data)

    def \_\_getitem\_\_(self, idx):

        example = self.data.iloc[idx]

        text = example["Content"]

        label = example["Label"]

        # Tokenize the text

        encoding = self.tokenizer.encode\_plus(text, padding='max\_length', truncation=True, max\_length=64, return\_tensors='pt')

        return {

            "input\_ids": encoding["input\_ids"].squeeze(0),

            "attention\_mask": encoding["attention\_mask"], ##.unsqueeze(0).int(),

            "label": label,

        }

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

A green bar graph with numbers

Description automatically generated

dataset\_train = HateSpeechDataset(train)

dataset\_val = HateSpeechDataset(validation)

#data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Create DataLoader

batch\_size = 128

dataloader\_train = DataLoader(

    dataset\_train,

    batch\_size=batch\_size,

    shuffle=True,

    num\_workers=2,

    collate\_fn=lambda x: {

        "input\_ids": torch.stack([item["input\_ids"] for item in x]),

        "attention\_mask": torch.stack([item["attention\_mask"] for item in x]),

        "labels": torch.tensor([item["label"] for item in x])

    },

    #collate\_fn=data\_collator,

    pin\_memory=True,

)

dataloader\_val = DataLoader(

    dataset\_val,

    batch\_size=batch\_size,

    shuffle=True,

    num\_workers=2,

    collate\_fn=lambda x: {

        "input\_ids": torch.stack([item["input\_ids"] for item in x]),

        "attention\_mask": torch.stack([item["attention\_mask"] for item in x]),

        "labels": torch.tensor([item["label"] for item in x])

    },

    #collate\_fn=data\_collator,

    pin\_memory=True,

)

# Define BERT classifier

class BERTClassifier(nn.Module):

    def \_\_init\_\_(self):

        # Specify network layers

        super(BERTClassifier, self).\_\_init\_\_()

        self.bert = BertModel.from\_pretrained('bert-base-uncased')

        self.avg\_pool = nn.AdaptiveAvgPool1d(1)

        self.linear = nn.Linear(self.bert.config.hidden\_size, 1)

        # Define dropout

        self.dropout = nn.Dropout(0.1)

        # Freeze BERT layers

        for n, p in self.bert.named\_parameters():

            p.requires\_grad = False

    def forward(self, text, masks):

        #output\_bert = self.bert(text, attention\_mask=masks).last\_hidden\_state.mean(axis=1)

        #print(output\_bert.last\_hidden\_state)

        #print(self.bert.config.hidden\_size)

        output\_bert = self.bert(text, attention\_mask=masks).last\_hidden\_state

        output\_bert = self.avg\_pool(output\_bert.transpose(1, 2)).squeeze(-1)

        return self.linear(self.dropout(output\_bert))

model = BERTClassifier()



# Define optimiser, objective function and epochs

optimizer = optim.Adam(model.parameters(), lr=0.001) #optim.AdamW(model.parameters(), lr=5e-5) #

criterion = nn.BCEWithLogitsLoss()

epochs = 5

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

## **Train with native Pytorch**

model.to(device)

val\_losses = []

train\_losses = []

# Train model

for epoch\_i in range(0, epochs):

    # ========================================

    #               Training

    # ========================================

    model.train()

    print(f"Start training epoch {epoch\_i}...")

    total\_train\_loss = 0

    for i, batch in enumerate(tqdm(dataloader\_train)):

        optimizer.zero\_grad()

        input\_ids = batch['input\_ids'].to(device)

        masks = batch['attention\_mask'].to(device)

        label = batch['labels'].to(device)

        output = model(input\_ids, masks)

        loss = criterion(output.squeeze(), label.float())

        loss.backward()

        # Clip the norm of the gradients to 1.0 to prevent the "exploding gradients".

        #torch.nn.utils.clip\_grad\_norm\_(model.parameters(), 1.0)

        optimizer.step()

        total\_train\_loss += loss.item()

    avg\_train\_loss = total\_train\_loss / len(dataloader\_train)

    train\_losses.append(avg\_train\_loss)

    # ========================================

    #               Validation

    # ========================================

    model.eval()

    print("Start validation...")

    y\_true\_bert = list()

    y\_pred\_bert = list()

    total\_eval\_loss = 0.0

    with torch.no\_grad():

        for batch in dataloader\_val:

            input\_ids = batch['input\_ids'].to(device)

            masks = batch['attention\_mask'].to(device)

            label = batch['labels'].to(device)

            output = model(input\_ids, masks)

            max\_output = (torch.sigmoid(output).cpu().numpy().reshape(-1)>= 0.5).astype(int)

            y\_true\_bert.extend(label.tolist())

            y\_pred\_bert.extend(max\_output.tolist())

            loss\_v = criterion(output.squeeze(), label.float())

            total\_eval\_loss += loss.item()

    avg\_val\_loss = total\_eval\_loss / len(dataloader\_val)

    val\_losses.append(avg\_val\_loss)

    print(f"Metrics after Epoch {epoch\_i}")

    print(f"Accuracy : {accuracy\_score(y\_true\_bert, y\_pred\_bert)}")

    print(f"Presision: {np.round(precision\_score(y\_true\_bert, y\_pred\_bert),3)}")

    print(f"Recall: {np.round(recall\_score(y\_true\_bert, y\_pred\_bert),3)}")

    print(f"F1: {np.round(f1\_score(y\_true\_bert, y\_pred\_bert),3)}")

    print("   ")

Start training epoch 0...

100%|██████████| 44/44 [00:11<00:00, 3.83it/s]

Start validation...

Metrics after Epoch 0

Accuracy : 0.7183333333333334

Presision: 0.731

Recall: 0.691

F1: 0.71

Start training epoch 1...

100%|██████████| 44/44 [00:10<00:00, 4.23it/s]

Start validation...

Metrics after Epoch 1

Accuracy : 0.7475

Presision: 0.741

Recall: 0.761

F1: 0.751

Start training epoch 2...

100%|██████████| 44/44 [00:10<00:00, 4.20it/s]

Start validation...

Metrics after Epoch 2

Accuracy : 0.7554166666666666

Presision: 0.754

Recall: 0.758

F1: 0.756

Start training epoch 3...

100%|██████████| 44/44 [00:10<00:00, 4.18it/s]

Start validation...

Metrics after Epoch 3

Accuracy : 0.7516666666666667

Presision: 0.761

Recall: 0.733

F1: 0.747

Start training epoch 4...

100%|██████████| 44/44 [00:10<00:00, 4.19it/s]

Start validation...

Metrics after Epoch 4

Accuracy : 0.7525

Presision: 0.765

Recall: 0.728

F1: 0.746

print('Test accuracy: {:.2f}'.format(accuracy\_score(y\_true\_bert, y\_pred\_bert)))

print('\nClassification report: \n', classification\_report(y\_true\_bert, y\_pred\_bert))

print('\nConfusion matrix: \n')

display(pd.DataFrame({"Predicted: Unhateful": confusion\_matrix(y\_true\_bert, y\_pred\_bert)[:, 0],

              "Predicted: Hateful": confusion\_matrix(y\_true\_bert, y\_pred\_bert)[:, 1]},

             index=['Actual: Unhateful', 'Actual: Hateful']))

Test accuracy: 0.75

Classification report:

precision recall f1-score support

0 0.74 0.78 0.76 1200

1 0.77 0.73 0.75 1200

accuracy 0.75 2400

macro avg 0.75 0.75 0.75 2400

weighted avg 0.75 0.75 0.75 2400

Confusion matrix:

|  |  |  |
| --- | --- | --- |
|  | Predicted: Unhateful | Predicted: Hateful |
| Actual: Unhateful | 932 | 268 |
| Actual: Hateful | 326 | 874 |

plt.figure(figsize=(10,5))

plt.title("Training and Validation Loss")

plt.plot(val\_losses,label="val")

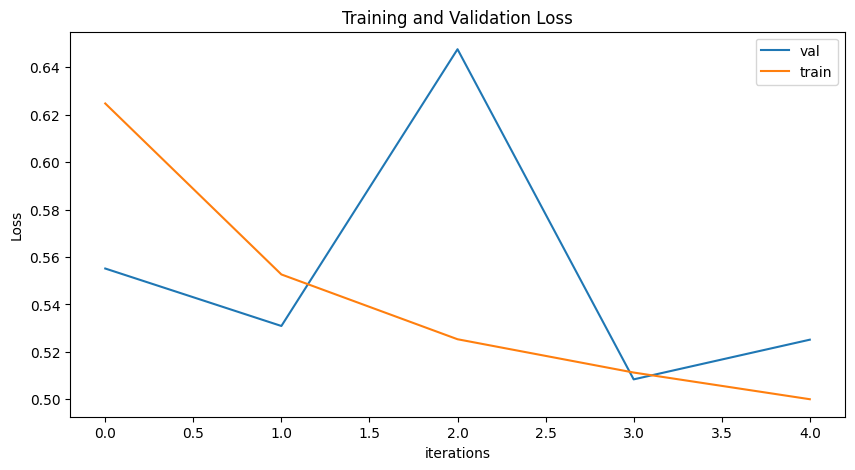
plt.plot(train\_losses,label="train")

plt.xlabel("iterations")

plt.ylabel("Loss")

plt.legend()

plt.show()



# **Fine-tuning a model with the Trainer API**[**¶**](https://www.kaggle.com/code/abramova/hate-speech-detection-from-tf-idf-to-transformers#Fine-tuning-a-model-with-the-Trainer-API)

In this section, along with Trainer API, I apply the AutoModelForSequenceClassification. This model automatically includes a classification head, so there's no need for any additional configuration. Train params are default.

model = AutoModelForSequenceClassification.from\_pretrained('bert-base-uncased',num\_labels=2)

training\_args = TrainingArguments(output\_dir="test\_trainer", report\_to="none")

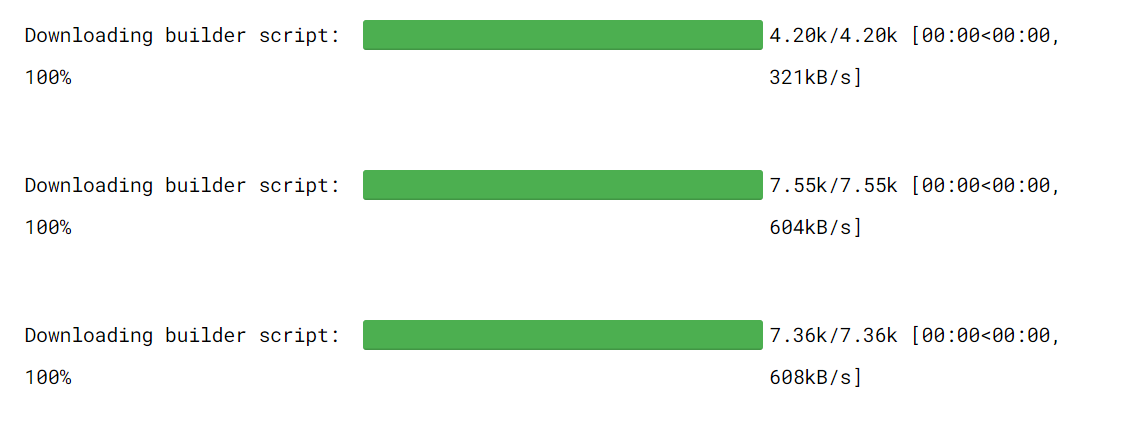
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

metric\_acc = evaluate.load("accuracy")

metric\_prec = evaluate.load("precision")

metric\_recall = evaluate.load("recall")



def compute\_metrics(eval\_pred):

    logits, labels = eval\_pred

    predictions = np.argmax(logits, axis=-1)

    acc = metric\_acc.compute(predictions=predictions, references=labels)["accuracy"]

    prec = metric\_prec.compute(predictions=predictions, references=labels)["precision"]

    rec = metric\_recall.compute(predictions=predictions, references=labels)["recall"]

    return {"accuracy": acc,"precision": prec, "recall": rec}

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

trainer = Trainer(

    model=model,

    args=training\_args,

    train\_dataset=dataset\_train,

    eval\_dataset=dataset\_val,

    compute\_metrics=compute\_metrics,

    data\_collator = data\_collator

)

trainer.train()



|  |  |
| --- | --- |
| Step | Training Loss |
| 500 | 0.533400 |
| 1000 | 0.356400 |
| 1500 | 0.275900 |
| 2000 | 0.134800 |

model.eval()

print("Start validation...")

y\_true\_auto\_bert  = list()

y\_pred\_auto\_bert = list()

total\_eval\_loss = 0.0

with torch.no\_grad():

    for batch in dataloader\_val:

        input\_ids = batch['input\_ids'].to(device)

        masks = batch['attention\_mask'].to(device)

        label = batch['labels'].to(device)

        output = model(input\_ids, masks)

        max\_output = np.argmax(output.logits.cpu().numpy(), axis=-1)

        y\_true\_auto\_bert.extend(label.tolist())

        y\_pred\_auto\_bert.extend(max\_output.tolist())

print(f"Accuracy : {accuracy\_score(y\_true\_auto\_bert, y\_pred\_auto\_bert)}")

print(f"Presision: {np.round(precision\_score(y\_true\_auto\_bert, y\_pred\_auto\_bert),3)}")

print(f"Recall: {np.round(recall\_score(y\_true\_auto\_bert, y\_pred\_auto\_bert),3)}")

print(f"F1: {np.round(f1\_score(y\_true\_auto\_bert, y\_pred\_auto\_bert),3)}")

print("   ")

print('Test accuracy: {:.2f}'.format(accuracy\_score(y\_true\_auto\_bert, y\_pred\_auto\_bert)))

print('\nClassification report: \n', classification\_report(y\_true\_auto\_bert, y\_pred\_auto\_bert))

print('\nConfusion matrix: \n')

display(pd.DataFrame({"Predicted: Unhateful": confusion\_matrix(y\_true\_auto\_bert, y\_pred\_auto\_bert)[:, 0],

              "Predicted: Hateful": confusion\_matrix(y\_true\_auto\_bert, y\_pred\_auto\_bert)[:, 1]},

             index=['Actual: Unhateful', 'Actual: Hateful']))

**DRIVE LINK:**

https://drive.google.com/drive/folders/11igXG3ZA1Wvuq3qLtTSyaThOrL-ftslF?usp=sharing