Hate Speech Detection - Data Cleaning

Kareem Ehab Kassab 900182771
Computer Science and Engineering
The American University in Cairo
kareemikassab@aucegypt.edu

Mahmoud Elshinawy 900183926
Computer Science and Engineering
The American University in Cairo
mahmoudelshenawy@aucegypt.edu

I. The Dataset

The data is obtained from Kaggle and it is the most recent as it updated on January 21st, 2022. It contains 135557 instances which is large enough. This dataset can be found here.

The data is very recent, and it has large enough size to train our model on. We will split our data into three sets Training, Validation, and Test sets. We used "fast_ml" to split the data into three datasets Training Set (80%) and Validation Set (10%) and Test Set (10%). Keeping that in mind, we have 108444 instances for training, 13556 for validation, and 13556 for testing. The data has 132 columns, none of which is of our importance but two columns, the actual text, and the label we are predicting which is whether the text is a hate speech or not.

```
In [134]:
def split_dataset(df, split_percent):
    return train_valid_test_split(df, target = 'hatespeech',\
                                  train_size=split_percent[0],\
                                  valid_size=split_percent[1],\
                                  test_size=split_percent[2])

In [136]:
X_train, y_train, X_valid, y_valid, X_test, y_test= split_dataset(df, [0.8,0

In [137]:
print(X_train.shape), print(y_train.shape)
print(X_valid.shape), print(y_valid.shape)
print(X_test.shape), print(y_test.shape)

(108444, 1)
(108444,)
(13556, 1)
(13556,)
(13556, 1)
(13556,)
```

Fig. 1. Split Data Code.

II. Data Cleaning and Feature Engineering

Data needs to be preprocessed before it is fed into any machine learning model. In this section, we are discussing the steps of data cleaning and feature engineering in our dataset. The steps includes punctuation removal, stopping word removal, and other steps to be explained in details in the next subsections.

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 135556 entries, 0 to 135555
Data columns (total 2 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   hatespeech  135556 non-null  float64
 1   text        135556 non-null  object
dtypes: float64(1), object(1)
memory usage: 2.1+ MB
```

Fig. 2. Split Data Code.

A. Importing Only Columns of Interest

The original dataset has 132 columns which is too much. After exploring all the columns, we decided to only keep two of them which are “text” and “hatespeech”. All the other 130 columns are not of importance for our model for the follwing reasons.

- “comment_id,annotator_id,platform”: all of these features does not indicate whether it is a hatespeech or not. We do not need our model to judge the commentor based on id, we need it to look only at the text written in the post regardless of who wrote it or which platform he/she uses.
- “sentiment, respect, insult, humiliate, status, dehumanize, violence, genocide, attack_defend, hate_speech_score”: All of these

are labels to be predicted by the model. We are only interested in “hatespeech” label.
- Other Columns: All the other columns are either target information or annotator information including race, religion and other attributes.

```
df= pd.read_csv("measuring_hate_speech.csv", usecols=["text", "hatespeech"])
df
```

4]:

Fig. 3. Code For Importing only the Two Columns.

B. Converting Our Classifier to Binary

At this point, our data contains a target label which can be one of three categories, HateSpeech, OffensiveLanguage, or neither. We decided to make our classifier binary by combining the first two categories into one for easier implementation as there is no much difference between the HateSpeech and OffensiveLanguage. Our classifier will classify the post as either "HateSpeech" in case it contains HateSpeech or OffensiveLanguage, and it will classify the post as "NotHateSpeech" in case the post contains neither. "0" represents "NotHateSpeech", "1" represents "HateSpeech"

```
df['hatespeech'].replace(2, 1, inplace=True)
df['hatespeech'].unique()
```

```
array([0., 1.])
```

Fig. 4. Converting Classifier into Binary.

C. Tokenization

Here, we split our text string into a list of words using regex. we are splitting words by any character that isn’t a list, a number, asterisc, at sign, and an exlamation mark. we decided to leave the last 3 special characters because they can be used to alter words that are still offensive, <b> for example:

- @sshole
- f*ck
- b!tch

the tokenizer converts all characters to lower case by default

```
def tokenize(text):
    split=re.split(r"[^A-Za-z0-9*@!]",text)
    return split
df['tokenized_text']=df['text'].apply(lambda x: tokenize(x.lower()))
df.head()
```

	hatespeech	text	tokenized_text
0	0.0	Yes indeed. She sort of reminds me of the elde...	[yes, indeed, , she, sort, of, reminds, me, of...
1	0.0	The trans women reading this tweet right now i...	[the, trans, women, reading, this, tweet, righ...
2	1.0	Question: These 4 broads who criticize America...	[question, , these, 4, broads, who, criticize,...

Fig. 5. Tokenization Function.

D. Removing Mentions, Punctuation, Extra White Spaces, Redundant Letters

For mentions and punctuation, We need to have the text clean without any special characters so we need to remove these characters except for the "!" character that may indicate offensive language. Before we remove the special characters, we also need to remove the mentions in the post as it does not help us indicate whether it is a hate speech or not, e.g. @myntwr. The code is in Fig.6.

```
def remove_mention_punctuation(textlist):
    #punctuation characters to be removed
    punc_except_exclamation= set(string.punctuation) - set('!')
    no_punct=[word for word in textlist if word not in list(punc_except_exclamat
    textstring= " ".join(no_punct)
    textstring= re.sub(r"@[A-Za-z0-9_]+"," ",textstring) #remove mention
    return textstring.split()
```

Fig. 6. Removing Mentions and Punctuation.

For extra white spaces and redundant letters,We also need to remove extra white spaces for cleaner data. Additionally we should detect if the post contains a word that has redundant letter. For instance if the user has typed "Fuck" as "Fuckkkkkkkkkkkk". The code is in Fig.7.

```
def remove_extra_characters(textlist):
    textstring= " ".join(textlist)
    textstring= re.sub(r"\s{2,}"," ",textstring)
    textstring= re.sub(r"(\w)\1{2,}", r"\1\1", textstring)
    return textstring.split()
```

Fig. 7. Removing Extra Spaces and Redundant Letters.

E. Converting Numbers from English Word to its Numerical Value

We also need to convert numbers from English words to its numerical value to be discarded later as it will not be of importance in detecting whether

the post is a hate speech or not. The code is in Fig.8.

```
def convert_english_numerical(textlist):
    textstring= " ".join(textlist)
    textstring= re.sub(r"([0-9]+)k",r"\g<1>000" ,textstring)
    textstring= re.sub(r"([0-9]+)M",r"\g<1>000000" ,textstring)
    textstring= re.sub(r"([0-9]+)B",r"\g<1>000000000" ,textstring)
    return textstring.split()
```

Fig. 8. Converting Numbers from English Word to its Numerical Value.

F. Converting Numbers from English Word to its Numerical Value and Removing Numbers

We also need to convert numbers from English words to its numerical value to be discarded later as it will not be of importance in detecting whether the post is a hate speech or not. The code is in Fig.9. At this stage, we should discard all numbers including the converted and the non-converted ones. The code is in Fig.10.

```
def convert_english_numerical(textlist):
    textstring= " ".join(textlist)
    textstring= re.sub(r"([0-9]+)k",r"\g<1>000" ,textstring)
    textstring= re.sub(r"([0-9]+)M",r"\g<1>000000" ,textstring)
    textstring= re.sub(r"([0-9]+)B",r"\g<1>000000000" ,textstring)
    return textstring.split()
```

Fig. 9. Converting Numbers from English Word to its Numerical Value.

```
def remove_numbers(text):
    # define the pattern to keep
    joined_text = " ".join(text)
    pattern = r'[^a-zA-Z.,!?/:;\"\'\s]'
    new_text = re.sub(pattern, '', joined_text)
    split_text = new_text.split(' ')
    return split_text

df['text_no_nums'] = df['text_no_html'].apply(lambda x: remove_numbers(x))
df.sample(n=10)
```

Fig. 10. Removing Numbers From the Data.

G. Removing Stop Words

Stop words are words such as you, you'd , he's , etc... We remove such words using the nltk library which has 179 stop words shown below. The function in fig.11. keeps only the text that is not in the stopwords list shown above, and adds the result to a new column.

H. Removing HTML Tags

Like punctuation, HTML tags are parts that are not needed in analyzing human language. they contribute in what we can call "noise" as they don't add much value in understanding and analysing text. We will use BeautifulSoup library for HTML tag clean-up. The code is in Fig.12.

```
def remove_stopwords(text):
    stopword = nltk.corpus.stopwords.words('english')
    text=[word for word in text if word not in stopword]
    return text
df['text_no_stopwords'] = df['tokenized_text'].apply(lambda x: remove_stopwords(
df.head()
```

	hatespeech	text	tokenized_text	text_no_stopwords
0	0.0	Yes indeed. She sort of reminds me of the elde...	[yes, indeed, , she, sort, of, reminds, me, of...	[yes, indeed, , sort, reminds, elder, lady, pl...
1	0.0	The trans women reading this tweet right now i...	[the, trans, women, reading, this, tweet, righ...	[trans, women, reading, tweet, right, beautiful]

Fig. 11. Removing Stop Words.

```
# function to remove HTML tags
def remove_html_tags(text):
    joined_text = " ".join(text)
    new_text= BeautifulSoup(joined_text, 'html.parser').get_text()
    split_text = new_text.split(' ')
    return split_text

df['text_no_html'] = df['text_no_accents'].apply(lambda x: remove_html_tags(x))
df.head()
```

Fig. 12. Removing HTML Tags.

I. Changing All Letters to Non-accented Letters

Converting accented letters such as: résumé, café, prótest, divorcé, coördinate, exposé, latté back to their original letters is important because most people using hate speech online can alter certain letters to not get detected. For example : retarded -> rétardéd The code is in Fig.13.

```
def remove_accented_chars(text):
    joined_text = " ".join(text)
    new_text = unicodedata.normalize('NFKD', joined_text).encode('ascii', 'ignor
    split_text = new_text.split(' ')
    return split_text

# call function
df['text_no_accents'] = df['text_no_stopwords'].apply(lambda x: remove_accented_
df.head()
```

Fig. 13. Changing All Letters to Non-accented Letters.

J. Expanding Contracted Words

Writing words in contracted format may lead to inconsistency, so we should expand all the contracted words for further cleaning, e.g. removing stop words. The code is in Fig.14.

K. Converting Words to Root (Lemmatization vs Stemming)

For word reduction and simplification There are two options: Stemming and Lemmatization. Our

```
def expand_contracted_words(textlist):
    textstring= " ".join(textlist)
    textstring= re.sub(r"\'s"," ",textstring)
    textstring= re.sub(r"\'ve"," have ",textstring)
    textstring= re.sub(r"\'ll"," will ",textstring)
    textstring= re.sub(r"\n't"," not ",textstring)
    textstring= re.sub(r"\'s"," ",textstring)
    textstring= re.sub(r"i'm"," i am ",textstring)
    textstring= re.sub(r"\'re"," are ",textstring)
    textstring= re.sub(r"\'d"," would ",textstring)
    return textstring.split()
```

Fig. 14. Expanding Contracted Words.

main aim here is to return words in forms such as the gerund to their "root" simple form to improve analysis accuracy and efficiency and make the whole process of NLP easier, simpler, and faster. This shall help the model generalize in order to find relevant results instead of one exact expression/form.

Stemming reduces a word to a form known as the word stem. The stem does not need to be a word that makes sense, there are many ways; usually, stemming cuts a word suffix without considering the context, for example: messages/messaging become messag. In cases that require context consideration or words to make sense it is not always appropriate to use, but it helps standardize text efficiently (faster than lemmatization).. The code for Lemmatization is in Fig.15.

```
import spacy
import en_core_web_sm

nlp = spacy.load('en_core_web_sm')
def get_lem(text):
    text= " ".join(text)
    text = nlp(text)
    text = ' '.join([word.lemma_ if word.lemma_ != '-PRON-' else word.text for w
    return text.split()
```

Fig. 15. Lemmatization Code.

L. Converting Text to Machine Readable Code (Word Embedding vs TF-IDF)

Word Embedding converts the word to a vector where similar words are mapped to the same vector and dissimilar words are not. Word embedding model is usually trained on a large data corpus, and is applied word by word. Having these vectors makes this step memory intensive.

TF-IDF stands for Term Frequency - Inverse Term Frequency. In short, the model will give higher weight to those words which did not appear that much and discard the common words. Unlike Word Embedding, TF-IDF maps each word to a single value not a vector. That would be also memory intensive to store this big matrix so they solved this problem by storing the word maps as sparse matrix where we store only the 1's in the matrix, and discarding the zeros. We are going to use TF-IDF to get the job done. The code is in fig.16.

```
dict_list= []
freq_dict={}
doc_dict={}
for doc in df['preprocess_data']:
    for item in doc:
        try:
            doc_dict[item]=doc_dict[item] + 1
        except:
            doc_dict[item]=1

        try:
            freq_dict[item]=freq_dict[item] + 1
        except:
            freq_dict[item]=1

    dict_list.append(doc_dict)
    doc_dict={}
```

Fig. 16. Word Frequency Code.

III. Feature Analysis

In this section, we will be discussing label distribution and feature correlation.

A. Classification Classes

The data has two classes 0 and 1. 0 represents "noHateSpeech", and 1 represents "HateSpeech". The data is a little bit biased towards the "noHateSpeech" class with 59.48 percent of the instances in the dataset, while the "HateSpeech" is 40.52 percent. The code for acquiring the pie chart is in fig.17. The pie chart for the class distribution is in fig.18.

```
count_not_hate= len(df[df['hatespeech']==0])
count_hate=len(df[df['hatespeech']==1])

pie_percentage= np.array([count_not_hate, count_hate])
Labels= ["NOT_HateSpeech", "HateSpeech"]
plt.pie(pie_percentage, labels = Labels, autopct='%1.2f%%')
plt.savefig("feature1.png")
plt.show()
```

Fig. 17. Code For Class Distribution.

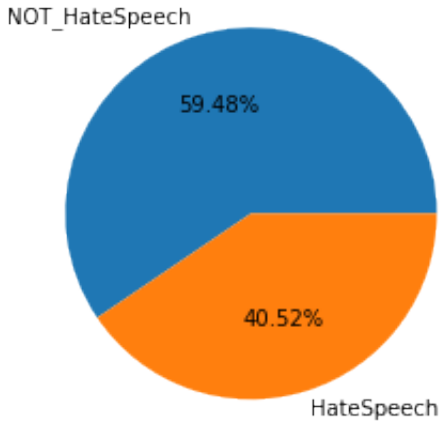

Fig. 18. Class Distribution.

B. Feature Selection

The dataset contains around 31011 unique words. We sorted the word dictionary in a descending way based on the frequency of that word in the dataset. The data initially had 96643 unique words. After the cleaning part including tokenization, removing stop words, punctuation, numbers, and other cleaning steps, it went down to 31011 for further cleaning process. We aim to only keep the words that are not very common, so we decided to keep the words with frequency 3 or less which is around 15000 words. The graph below is showing a plot of 5000 most common words. The plot is in fig.18.

C. Feature Correlation

In this subsection we will investigate the correlation between the words we kept and the class label associated with them. Given the fact that we keep only the words with frequency more than or equal to 3, drawing the correlation between the label and the words will not add much information about associating the remaining words with the labels

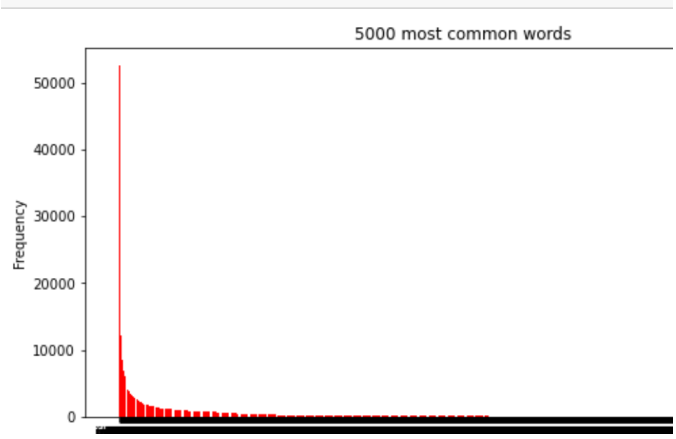

Fig. 19. 5000 Most Common Words.