Offensive Language and Hate Speech Detection Using Sentiment Analysis

Jovana Zdravkovska[1], Magdalena Petrusevska[1] and Sara Paunkoska[1]

[1]University Ss Cyril and Methodius in Skopje, Faculty of Computer Science and Engineering

Abstract. Sentiment algorithms can be utilized to evaluate and identify hate speech and offensive language, enabling them to be filtered out from online posts. Using a predefined dataset and rules for said datasets, many algorithms can be trained to recognize patterns of words that can be deemed inappropriate for online use and flagged as hate speech. Such algorithms are Naive Bayes, Support Vector Machines, LSTM, Bert and many more. The general idea of all those algorithms is the same, to take the given text, dig deep inside and detect the intent of it. But to be able to achieve our goal, we first need to define what the problem that we are trying to solve is and how it affects those targeted.

Keywords. Hate Speech, Offensive Language, Sentiment, Tweets, Convolutional Neural Networks, Long Short-Term Memory Networks, XGBoost Algorithm, BERT, K-Means Clustering, Agglomerative Clustering.

1. Introduction

Humans are not strangers to using offensive language in their day to day lives. Even some would say that its part of being human. And as with everything else humans have invented, their actions have web itself in those inventions. Internet technologies are no such exception. Despite having huge benefits for humans, the enormous abundance of information, connecting people from all over the world from different cultures and religions, as well as entertainment in all kinds of different forms, the careless and inconsiderate usage of the technologies can bring as many disadvantages to people as it brings advantages. One such disadvantage is the amount of hate people spread on all platforms. Hiding behind their usernames and their anonymity, people take full advantage of the shadow they obtain with it. To say all the things, they otherwise might not have been able to say in person. Having little to no repercussion for their actions, people stop filtering their words and stop carrying how their words affect those to whom they are directed. With that in mind methods for assessing and filtering what is posted online are more than necessary.

2. Background

As mentioned before people tend to express themselves using explicit language, which by itself is not so harmful. However, combined with the anonymity the internet provides and absence of serious repercussion, people take it to the extreme, targeting specific groups of people, races, nationalities and more.

2.1. Hate Speech

But when does offensive language and statements turn to hate speech? Easy, when those statements are directed towards a group of people, usually but not necessarily minorities, that discriminates, degrades, or otherwise reticulates against them based on that group race, gender, ethnicity, nationality, etc. Even though hate speech is not a new or recent problem, in the words of UN secretary – general ANTÓNIO GUTERRES, “Social media provides a global megaphone for hate”.[1] As mentioned people are not new to spreading hate about one another, so why does this

behavior present such a threat when done online? Simply put it's because when done online its ever so easier to spread misinformation, that unlike traditional media can reach global audience in real time.[2] This spread of hate speech across the world is resulting in increase in violence towards the targeted groups.[3]

2.2. Cyberbullying

Much like hate speech, the use of the anonymity that the internet provides a rise in cyber bullying can be noticed. What separates cyber bullying from hate speech is that instead of the hate comments being directed towards a group of people, they are directed towards one specific person. With how widespread the internet is the harassment and pressure of posts targeted towards the victims can feel constant and never-ending. Making the recipients feel like there is no escape and resulting in emotional, mental and/or physical damage to the individual.[4]

3. Datasets

In this section we give a comprehensive overview of the data that makes up the dataset used in this research. We will give a quick overview of the data preparation process, as well as better understanding of patterns within the data, detect outliers or anomalous events.

3.1. Data overview

This dataset was assembled using the crowd-sourcing method to label a sample of tweet data into three categories: those containing hate speech, only offensive language, and those with neither. In order to distinguish between these different categories, the algorithms were trained using a multi-class classifier. It should be emphasized that this dataset includes text that may be deemed discriminatory, derogatory, or offensive based on race, gender, sexual orientation, or other factors.

3.2. Data preparation

To make raw data ready for further processing and analysis, we decided to cut down the other columns that were not of high importance. This results in using only 2 columns: 'Class' and 'Tweet', where the first one contains the label of the actual text stored in the second column. Then it was important to check if we were dealing with missing values. In pursuit of this aim we made a quick calculation that showed us that there are no missing values (see Table. 1).

```
missing_values = df.isnull().sum()
percentage = 100 * df.isnull().sum() / len(df)
missing_values_table = pd.concat([missing_values, percentage], axis=1)
missing_values_table.columns = ['Num. of missing values','% of missing values']
missing_values_table
```

Table 1. Calculation for missing values

	Num. of missing values	% of missing values
class	0	0.0
tweet	0	0.0

Further to ensure that we are not dealing with irrelevant data from a raw dataset, tweet cleaning process took place. Using a function that checked the relevance of the data was used.

```
def clean_text(text):
    text = str(text).lower()
    text = re.sub('\[.*?\]', '', text)
    text = re.sub('https?://\S+|www\.\S+', '', text)
    text = re.sub('<.*?>+', '', text)
```

```
text = re.sub('[%s]' % re.escape(string.punctuation), '', text)
text = re.sub('\n', '', text)
text = re.sub('\w*\d\w*', '', text)
text = re.sub('rt ','',text)
text = [word for word in text.split(' ') if word not in stopword]
text=" ".join(text)
text = [stemmer.stem(word) for word in text.split(' ')]
text=" ".join(text)
return text
```

3.3. Exploratory data analysis

Some characteristics for the dataset, that we get to know with the help of EDA is that the dataset consist total of 24783 entries. Using predefined methods a summary statistic of the dataset can be obtained (see Table. 2).

```
df.describe().T
```

Table. 2. Summary statistics for the dataset

	count	mean	std	min	25%	50%	75%	max
class	24783.0	1.110277	0.462089	0.0	1.0	1.0	1.0	2.0

While calculating the total of each of the samples by class label, we can see that the data in this dataset is unbalanced (see Table. 3 and Fig. 1). Balancing a dataset makes training a model easier because it helps prevent the model from becoming biased towards one class. We don't want our model to favor the majority class just because it contains more data.

```
df['class'].value_counts(normalize=True)
```

Table. 3. Unbalanced result of the dataset

1	0.774321
2	0.167978
0	0.057701
Name: class,	dtype: float64

```
df['class'].hist()
plt.xticks(rotation = 90)
df['class'].hist(bins = 3)
```

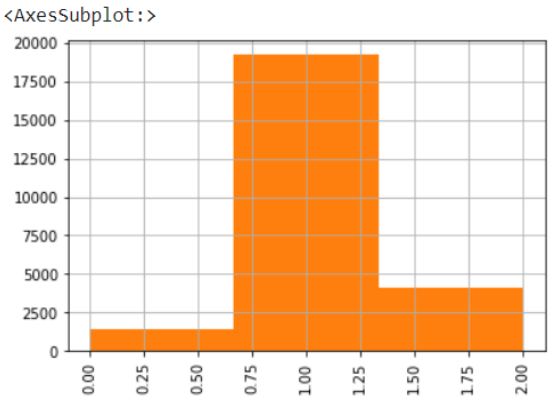

Fig. 1. Graph of the unbalanced dataset

3.4. Data balancing

The data balancing technique that was used to overcome this problem is RandomOverSampler. This technique is used when the minority class is underrepresented in the dataset, and the goal is to increase the number of samples in the minority class.

```
from imblearn.over_sampling import RandomOverSampler
import numpy as np
X = df.drop('class', axis=1)
y = df['class']
ros = RandomOverSampler(random_state=0)
X_resampled, y_resampled = ros.fit_resample(X, y)
unique, counts = np.unique(y_resampled, return_counts=True)
print(dict(zip(unique, counts)))
```

3.5. Privacy concerns

The data used in this dataset is expected to comply with the General Data Protection Regulations in Europe (GDPR) and address growing concerns over online privacy, since it does not uncover the privacy of the authors of the comments that were used. Any personally identifiable information, such as individuals' names (excluding celebrity names), has been redacted.

4. Models

Our work explores different models to compare and contrast various approaches to the task. Firstly, we introduce CNN+LSTM, which is a neural network architecture that combines convolutional neural networks (CNNs) and long short-term memory (LSTM) networks. XGBoost was the second algorithm we employed in our analysis, which works by iteratively improving the predictions of a weak model by adding new models that minimize the residual error of the previous models. And the last one is the BERT, which is pre-trained language model developed by Google for natural language processing (NLP) tasks.

The following sections describe these model architectures in detail, the algorithms they are based on, and the features they use.

4.1. CNN + LSTM

The integration of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks presents a potent and versatile method for deep learning that is applicable to various domains and use cases. Convolutions don't use fully connected layers, but sparsely connected layers, that is, they accept matrices as inputs. Unlike human beings, who understand images by taking snapshots with their eyes. The layers used in CNNs are called:

- Convolutional layer
- Pooling layer
- Fully connected layer

LSTMs on the other hand, make small modifications to the information by multiplications and additions. With LSTMs, the information flows through a mechanism known as cell states. This way, LSTMs can selectively remember or forget things. The information at a particular cell state has three different dependencies. These dependencies can be generalized to any problem as:

- The previous cell state (i.e., the information that was present in the memory after the previous time step)
- The previous hidden state (i.e., this is the same as the output of the previous cell)
- The input at the current time step (i.e., the new information that is being fed in at that moment)

In this study we chose to build our model using the following layer structure:

1.**Embedding layer**: This layer creates an embedding matrix of size (max_features, 30) where each feature vector has a length of 30. The input_length is the number of time steps or the length of the input sequence.

2.**Conv1D layer**: This is a one-dimensional convolutional layer that applies 30 filters of size 3 to the input sequence. The padding parameter is set to 'same', which means that the output sequence has the same length as the input sequence. The activation function used is ReLU.

3.**MaxPooling1D layer**: This layer performs max pooling over a pool size of 2, reducing the length of the output sequence by a factor of 2.

4.**Another Conv1D layer**: This layer applies 20 filters of size 3 to the output of the previous layer.

5.**Another MaxPooling1D layer**: This layer performs max pooling over a pool size of 2 again.

6.**Another Conv1D layer**: This layer applies 10 filters of size 3 to the output of the previous layer.

7.**Another MaxPooling1D layer**: This layer performs max pooling over a pool size of 2.

8.**LSTM layer**: This layer has 100 units and uses dropout and recurrent dropout with a rate of 0.2 to prevent overfitting. LSTM is a type of recurrent neural network that can handle sequential data.

9.**Dense layer**: This is the output layer with num_classes units and a softmax activation function. The model predicts the probability of each class given the input sequence.

```
import keras.backend as K
from keras.models import Sequential
from keras.layers import Dense,Embedding,Conv1D,MaxPooling1D,LSTM, Flatten
from sklearn.metrics import accuracy_score,confusion_matrix,classification_report
K.clear_session()
model = Sequential()
model.add(Embedding(max_features, 30, input_length=X_train.shape[1]))
model.add(Conv1D(filters=30, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Conv1D(filters=20, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Conv1D(filters=10, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(num_classes, activation='softmax'))
```

Overall, this model uses a combination of convolutional and recurrent layers to extract features from the input sequence, and then uses a fully connected layer for classification.

Then the model is trained using the Adam optimizer and the categorical cross-entropy loss. To train the model, we use Adam optimizer, which can dynamically adapt the learning rate and categorical cross-entropy loss, which is a

common loss function for multiclass classification problems. And the evaluation metric that was used is the accuracy metric.

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

The last part, fitting the model, we fitted the compiled model to the training data (**X_train** and **Y_train**) for a specified number of epochs (**epochs=15**) with a batch size of 130 (**batch_size=130**).

Here's a breakdown of what each parameter means:

- **X_train**: This parameter specifies the input data for training the model.
- **Y_train**: This parameter specifies the target or label data for training the model.
- **epochs=15**: This parameter specifies the number of times the model will be trained on the entire dataset.
- **batch_size=130**: This parameter specifies the number of samples that will be used in each batch during training.
- **verbose=2**: This parameter specifies the level of detail to be printed during training. **verbose=2** means that progress updates will be printed to the console at the end of each epoch.
- **validation_split=0.2**: This parameter specifies the percentage of training data that will be used for validation during training.

The output of **model.fit()** is an object called **model_history**, which contains information about the training history of the model, such as the loss and accuracy for each epoch during training.

4.2. XGBoost

XGBoost is based on the idea of creating an ensemble of decision trees and iteratively improving their predictions. The basic idea of XGBoost is to create a set of decision trees, where each tree learns to predict the residual error (i.e., the difference between the actual target value and the predicted value) of the previous tree. This process is repeated iteratively, with each new tree added to the ensemble helping to reduce the residual error of the previous trees. The following screenshot reveals the model structure used in our study.

```
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split( data[data.columns[:]],  df['class'], test_size=0.2)
from xgboost import XGBClassifier
#inicialize XGBoost
model = XGBClassifier(max_depth=60, min_child_weight=1,  n_estimators=250,n_jobs=1 , verbose=1,learning_rate=0.16)
# Add silent=True to avoid printing out updates with each cycle
model.fit(X_train, Y_train)
#make prediction
y_pred = model.predict(X_test)
```

XGBClassifier(max_depth=60, min_child_weight=1, n_estimators=250, n_jobs=1, verbose=1, learning_rate=0.16): This line initializes an instance of the XGBClassifier with several hyper parameters that control the model's behavior:

- max_depth=60: The maximum depth of the decision trees. Increasing this value may improve the model's performance, but also increases the risk of overfitting.
- min_child_weight=1: The minimum weight required for a child node to continue splitting. This parameter helps to prevent overfitting by controlling the minimum number of samples required to make a split.
- n_estimators=250: The number of trees in the model. Increasing this value may improve the model's performance, but also increases the computational cost.
- n_jobs=1: The number of parallel threads to use for training the model. In this case, only one thread is used.

- verbose=1: The verbosity level of the training output. In this case, the model will print progress updates for each tree.
- learning_rate=0.16: The learning rate of the model, which controls the step size at each iteration. A higher learning rate will cause the model to converge more quickly, but may also make it more likely to get stuck in suboptimal solutions.

model.fit(X_train, Y_train): This line trains the XGBoost model using the input data (X_train) and corresponding labels (Y_train).

y_pred = model.predict(X_test): This line generates predictions for the test data (X_test) using the trained model.

At the end the accuracy_score() function from the metrics module is used to compute the accuracy.

4.3. Bert

Another machine learning model we use to recognize hate speech and offensive language in our selected tweets dataset is BERT (Bidirectional Encoder Representations from Transformers). More specifically, we exploit the ability of BERT as a pre-trained model for establishing context between textual data and direct the model to meet our need for marking tweets that contain hateful words and phrases.

To enable the model to do more efficient analysis of the data we feed it with, we work only with two columns of data: class and tweet. In the class column numerical values have been stored with the following clarification:

- Hate speech --> 0
- Offensive --> 1
- Neither ---> 2

An important aspect of the tweets dataset we work with is that it is not balanced and the class with which we denote the presence of offensive language, i.e. class 1 is dominant. To prevent BERT from becoming biased towards class 1, it is of great importance for us doing balancing of the dataset.

As with any previous deep learning representation, we perform train-test split procedure on the re-sampled data in 80:20 ratio. The split allows us to train our model on one of the newly created datasets and later test its accuracy and overall performance.

```
from sklearn.model_selection import train_test_split
X_train,X_test,Y_train,Y_test = train_test_split(X_resampled,y_resampled,test_size=0.2)
```

The data we have split is stored in two separate datasets: df_bert_train and df_bert_test. In addition, we rename the labels of the columns we work with to ensure that the fine-tuned BERT is trained with the correct labels for the specific task of classifying tweets.

For sequence classification tasks such as ours, we initialize a pre-trained BERT model called bert-base-uncased meaning it uses an uncased vocabulary. With AutoTokenizer class, a tokenizer for the specific model is selected automatically.

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
```

We load previously saved datasets as one dataset to which we apply a map function to tokenize each row of the Tweet column. The process of tokenization involves converting individual words into numeric IDs that can be delivered to a neural network for further processing. For clarification, the truncation=True argument means that tweets longer than the maximum length which the existing BERT model originally handles will be cut to fit its capabilities. The batched=True argument dictates that the tokenization should be operated on batches of data which will speed the whole process when working with large datasets such as our tweets dataset. Lastly, the load_from_cache_file=False specifies a restriction for loading the dataset from cache and rather applying the tokenizer function directly to the encoded_dataset.

```
encoded_dataset = dataset.map(lambda t: tokenizer(t['Tweet'],  truncation=True), batched=True,load_from_cache_fi
le=False)
encoded_dataset
```

We build our own model using a class from the transformers package called AutoModelForSequenceClassification and apply from_pretrained as a method that initializes an instance of the pre-trained BERT model. Once the model is initialized, we do fine-tuning for a task with three output classes which is emphasized by the num_labels=3 argument.

```
model = AutoModelForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=3)
```

Training our model involves invoking the train() method on the trainer object. This object is an instance of the Trainer class which takes the pre-trained model, some training arguments that are defined by us, the appropriate tokenizer and the training and evaluation datasets. The training arguments imply a defined learning rate, number of training epochs, the batch sizes for the training and evaluation datasets as well as the name of the output directory for saving the model. For argument values we took numbers that gave good results in other similar models.

```
from transformers import TrainingArguments, Trainer
arg = TrainingArguments(
"sentiment",
learning_rate=5e-5,
num_train_epochs=1,
per_device_eval_batch_size=8,
per_device_train_batch_size=8,
seed=19
)
trainer = Trainer(
model=model,
args=arg,
tokenizer=tokenizer,
train_dataset=encoded_dataset['train'],
eval_dataset=encoded_dataset['test']
)
trainer.train()
```

When we finally have a finished model in our hands it is time to test it on the encoded evaluation dataset.

```
y_pred = trainer.predict(encoded_dataset['test'])
```

This line results in another object called PredictionOutput and that is why it is needed to assign the predictions attribute of this object to the y_pred variable.

```
y_pred = y_pred.predictions
```

But to enable the resulting variable to produce a list of predicted sentiment labels for each tweet sample from the evaluation dataset we select the index and connect it to the highest score we get by using the argmax() function from the NumPy library:

```
y_pred = [np.argmax(y_pred[i]) for i in range(0,len(y_pred))]
```

4.4. Clustering

Clustering is a popular machine learning technique that refers to the process of grouping similar data points into groups or so-called clusters. In the context of tweet data, we use clustering to identify all tweets that are internally related to each other based on the sentiment they have. It is important to note that whichever clustering algorithm is used, the number of clusters obtained on our hate speech detection dataset should be three. The second important thing to note is that the algorithms do not work with string data. For that specific reason we perform embedding the string data in our tweet column:

```
message_embeddings = embed(df['tweet'].values)
```

4.5. K—Means clustering

One type of an unsupervised machine learning algorithm that we use for clustering the tweets data is K-Means clustering. To start with, we create an instance of the algorithm and specify the number of clusters:

```
km = KMeans(n_clusters=3)
```

For our provided dataset there are three because of the previously defined classes for sentiment analysis. Next, we use the fit method of the object we created to apply the data which we want to cluster and start the algorithm:

```
km.fit(message_embeddings)
```

For the data we have, it is meaningful to do some dimensionality reduction. We use another algorithm called PCA and specify that we want to reduce the dimensionality of the data to two dimensions. With the fit_transform method we perform the reduction on our data:

```
data = message_embeddings
pca = PCA(2)
#Transform the data
df_clustering_new = pca.fit_transform(data)
```

Once again, we apply the K-Means algorithm, but this time on the transformed data using the fit_predict method. The results of the clusters we got are stored in a new variable called label. Finally, labels_ attribute is used to retrieve the cluster labels of the original data we provided message_embeddings after that data has been transformed in 2D and clustered with K-Means algorithm for clustering.

```
#Initialize the class object
kmeans = KMeans(n_clusters=3)
#predict the labels of clusters.
label = kmeans.fit_predict(df_clustering_new)
#Getting unique labels
u_labels = np.unique(label)

labels = kmeans.labels_
```

The visualization of the clustering results indicates that the data has been clustered into three clusters as we told the algorithm and because of the fact that our dataset has been balanced before meaning that each class has same number of labels, we got the following scatter plot (see Fig.2):

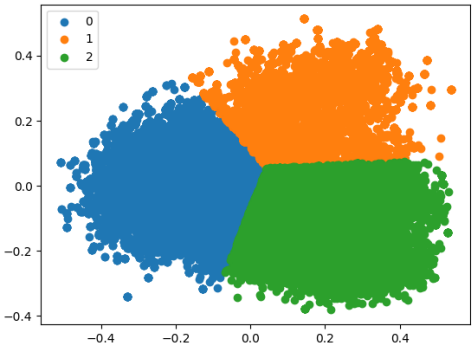

Fig. 2. K-Means clustering result

4.6. Agglomerative clustering

Agglomerative clustering is another type of algorithm we use to cluster the data for hate speech and offensive language detection. The method it uses is starting with each data point as a separate cluster and then merging the two closest clusters found at each next step. Similar as with the K-Means cluster, we first make the data two dimensional and then define the data as a graph structure where each data point is connected to its ten nearest neighbors using the distance metrics:

```
connectivity = kneighbors_graph(df_new, n_neighbors=10, include_self=False)
```

Actually, in the background the graph is represented as a sparse matrix which we pass to the AgglomerativeClustering function as an argument.

```
model = AgglomerativeClustering(n_clusters=3, connectivity=connectivity, linkage='ward').fit(df_new)
```

With this command we create an object which stores the results after proceeding to perform the clustering on the input data df_new making the algorithm create three clusters and use specific linkage criterion ward which minimizes the variance of the clusters being merged. Same as with the other algorithm we explained, the labels_ attribute of the model object is accessed to obtain all of the cluster labels and to be converted to a list:

```
clusters = model.labels_.tolist()
```

The visualization we get is clear and easy to interpret. We can see the borders between each of the clusters and summarize that the class number 2 has the smallest cluster (see Fig. 3):

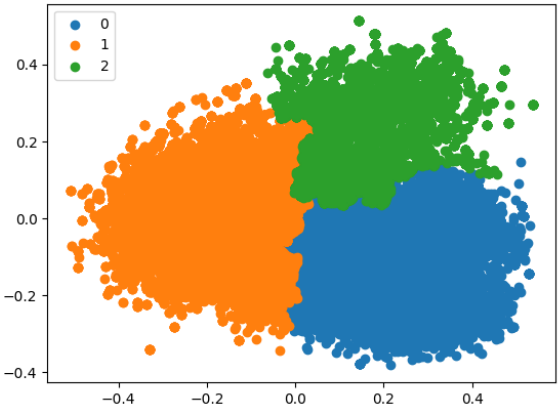

Fig. 3. Agglomerative clustering result

5. Results and discussion

Initially, we chose BERT as it is popular for its state-of-the-art performances on a variety of natural language processing tasks including sentiment analysis under which falls our research. Specifically, the model we created using a pre-trained BERT model gave enviable numbers for most of the metrics we have measured. For our model, we've got the following classification report (see Table.4):

Table.4. Classification report results

	precision	recall	f1-score	support
0	0.95	0.99	0.97	3900
1	0.97	0.92	0.95	3801
2	0.98	0.99	0.98	3813
accuracy			0.97	11514
macro avg	0.97	0.97	0.97	11514
weighted avg	0.97	0.97	0.97	11514

Each of the three classes got high precision numbers [0.95, 0.97, and 0.98] which indicates that the model makes few false positive predictions. Similarly, the model gave even higher recall numbers [0.99, 0.92 and 0.99]. This shows that the model makes even less false negative predictions. To balance the precision and recall results we look at another metric called f1-score. It is important that we do not just rely on the overall accuracy of the model and take f1-score as a metric that we trust the most. Lastly, support as a metric gives us the number of instances in each class of the data.

6. Analysis

To be able minimize the extreme hatred present in the online world today, in this paper we demonstrate one way to start overcoming this problem. Using the steps outlined in the paper we can obtain algorithms that can accurately measure the intent of the text that has been posted online. With the right use of these algorithms and methods we can obtain sufficient filters and validation techniques to effectively minimize the hatred that is spread online.

7. Conclusion

With the swift expansion of social media and internet technologies, and all the more younger users of it, it is necessary to develop methods to reduce and limit the spreading of all the hate speech online. With this we can help decrease the violence and discrimination towards minority and vulnerable groups of people as well as individuals. In this paper we can investigate the first steps that need to be taken to make the internet a safer place for everyone using it.

8. Acknowledgements

A big appreciation to all the members of the team that contributed to the research and execution of the project, as well as our coordinator and mentor Georgina Mirceva that guided us through the entirety of the project and its research.

References:

[1][2] https://www.un.org/en/hate-speech/understanding-hate-speech/what-is-hate-speech#:~:text=Social%20media%20provides,Secretary%2DGeneral%2C%202021 [25.03.2023]
[3] https://www.cfr.org/backgrounder/hate-speech-social-media-global-comparisons#:~:text=Hate%20speech%20online%20has%20been%20linked%20to%20a%20global%20increase%20in%20violence%20toward%20minorities%2C%20including%20mass%20shootings%2C%20lynchings%2C%20and%20ethnic%20cleansing. [25.03.2023]
[4] https://www.unicef.org/end-violence/how-to-stop-cyberbullying [25.03.2023]