Final Project Submission

- Student name: Sabina Bains
- Student pace: Self Paced
- Scheduled project review date / time: 8/2 11am EST
- Instructor name: Claude Fried

Business Understanding

Over the past couple of months, Twitter has been in the headlines for the potential acquistion of their company from Elon Musk. This has upset many people, as Musk plans to reduce the level of moderation Twitter currently has on potentially harmful tweets.

BeKind Org. believes this change will likely lead to a massive increase in hate speech and misinformation, and will negatively affect users. They therefore would like our help to create the model behind a browser extension that can flag tweets that are considered cyberbullying. This way, users have the option to hide tweets from their feed.

We will tackle this with a supervised learning approach, therefore training our models a [dataset (https://www.kaggle.com/code/anayad/classifying-cyberbullying-tweets/data)](https://www.kaggle.com/code/anayad/classifying-cyberbullying-tweets/data) of 47K tweets that have already been flagged for cyberbullying type. Since this analysis involves natural language processing, we will have to process our data accordingly.

We will first train a multinomial naive bayes model, then compare our model's accuracy with recurrent neural networks to see which performs better on unseen data. The final model will then be used in the extension to sift through tweets and flag by type of cyberbullying.

Data Understanding

Reading in Necessary Packages

In [1]:

```
# importing standard packages
import pandas as pd
import re
import numpy as np

# importing packagest for NLP and Multinomial Naive Bayes
from nltk.tokenize import RegexpTokenizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import cross_val_score, train_test_split
from nltk import FreqDist
from nltk.corpus import stopwords
from nltk.stem.snowball import SnowballStemmer
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.util import ngrams

# importing packages for plotting results
from sklearn.metrics import plot_confusion_matrix, ConfusionMatrixDis
import matplotlib.pyplot as plt
import matplotlib as mpl

# changing colors of output
COLOR = 'white'
mpl.rcParams['text.color'] = COLOR
mpl.rcParams['axes.labelcolor'] = COLOR
mpl.rcParams['xtick.color'] = COLOR
mpl.rcParams['ytick.color'] = COLOR


# importing keras packages for neural networks
import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Input, Dense, LSTM, Embedding, Dropout, Acti
from keras import initializers, regularizers, constraints, optimizers
from keras.preprocessing import text, sequence

# importing warning package to ignore
import warnings
warnings.filterwarnings('ignore')
```

Reading in Data

In [2]:

```
# reading in data
df = pd.read_csv('data/cyberbullying_tweets.csv')


# replacing offensive words
df.tweet_text = df.tweet_text.str.replace('ggers','xxxxx')
df.tweet_text = df.tweet_text.str.replace('gger','xxxx')
```

Understanding Labels

In [4]:

```
# checking distribution of target values - looks like a balanced data
pd.DataFrame(df.cyberbullying_type.value_counts(normalize=True))
```

	cyberbullying_type
religion	0.167701
age	0.167575
gender	0.167177
ethnicity	0.166925
not_cyberbullying	0.166590
other_cyberbullying	0.164032

In [5]:

```
# standardizing words to all be lowercase
def lowercase(x):
    return x.lower()

df.tweet_text = df.tweet_text.apply(lowercase)
```

In [6]:

```
# tokenizing words in df to inspect differences in cyberbullying clas
def tokenize(x):
    token_pattern = r"(?u)\b\w\w+\b"
    tokenizer = RegexpTokenizer(token_pattern)
    return tokenizer.tokenize(x)

df['tokens'] = df.tweet_text.apply(tokenize)
```

In [7]:

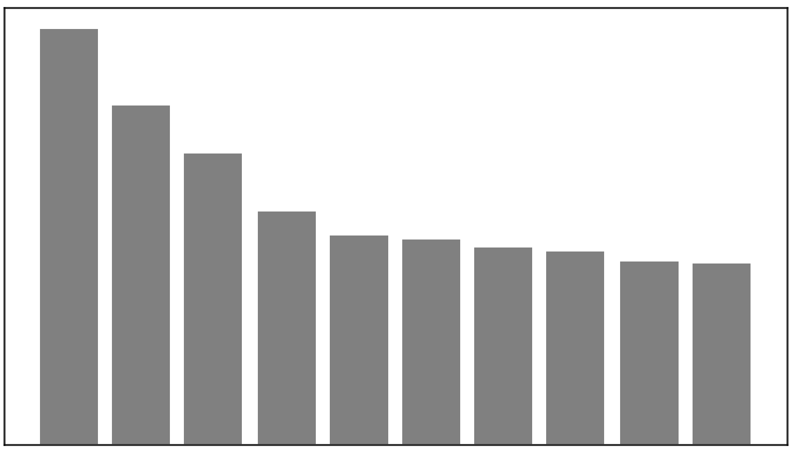
```
# creating function to remove stopwords
stopwords_list = stopwords.words('english')

def remove_stopwords(token_list):
    return [word for word in token_list if word not in stopwords_list

df.tokens = df.tokens.apply(remove_stopwords)
```

In [8]:

```
# Above we can see we need to stem the data (muslim vs. muslims). cre
stemmer = SnowballStemmer(language="english")

def stem(tweets):
    return [stemmer.stem(tweet) for tweet in tweets]

df.tokens = df.tokens.apply(stem)
```

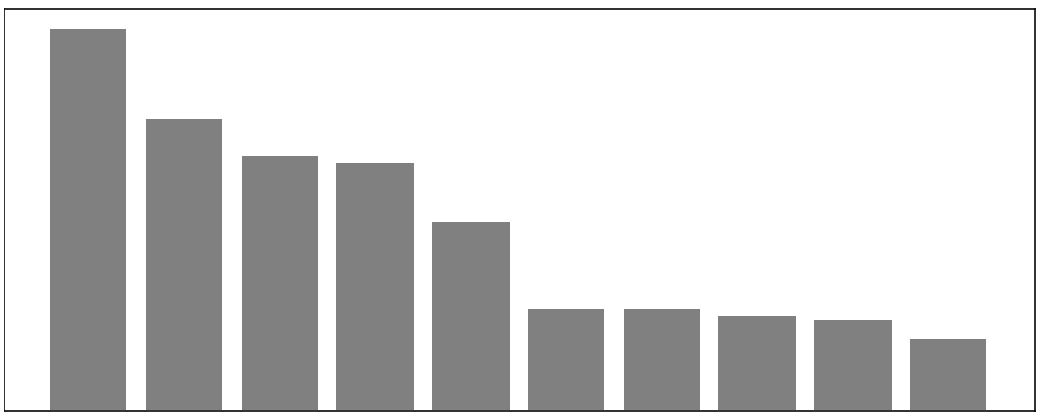
In [110]:

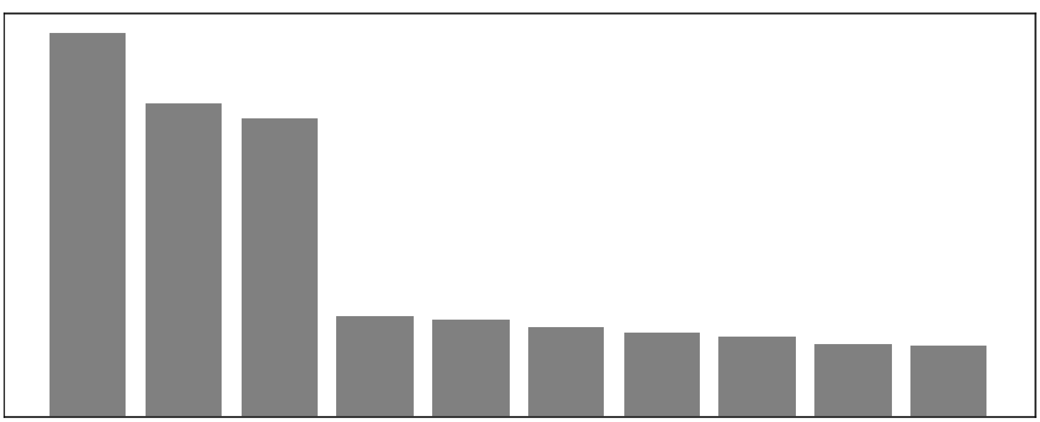
```
# looking at top words in entire dataset, regardless of label
total_top_10 = FreqDist(df.tokens.explode()).most_common(10)

tokens = [word[0] for word in total_top_10]
counts = [value[1] for value in total_top_10]

fig = plt.figure(figsize=(7,4))
plt.bar(tokens,counts, color= 'grey')
plt.title('Top Words')
plt.xlabel('Words')
plt.ylabel('Count')
plt.show()
```

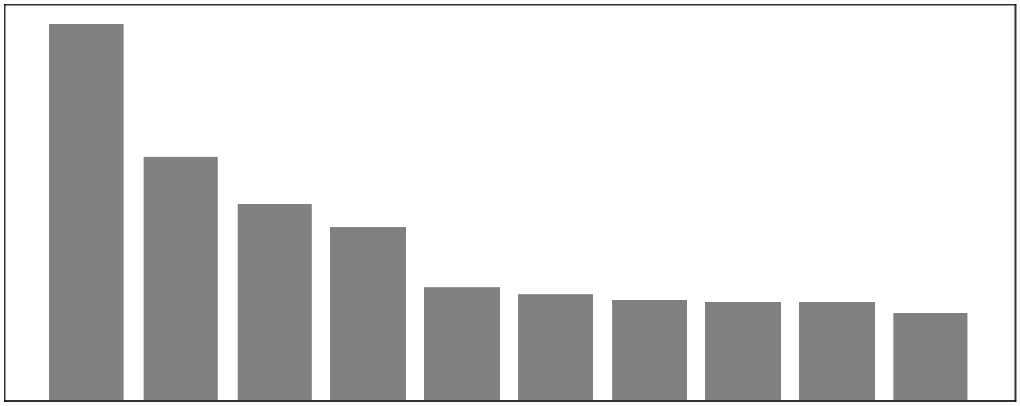


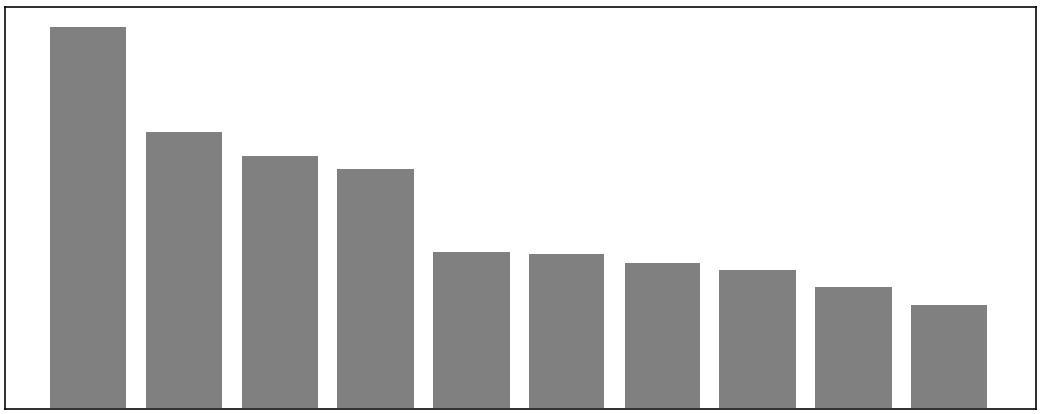
In [10]:

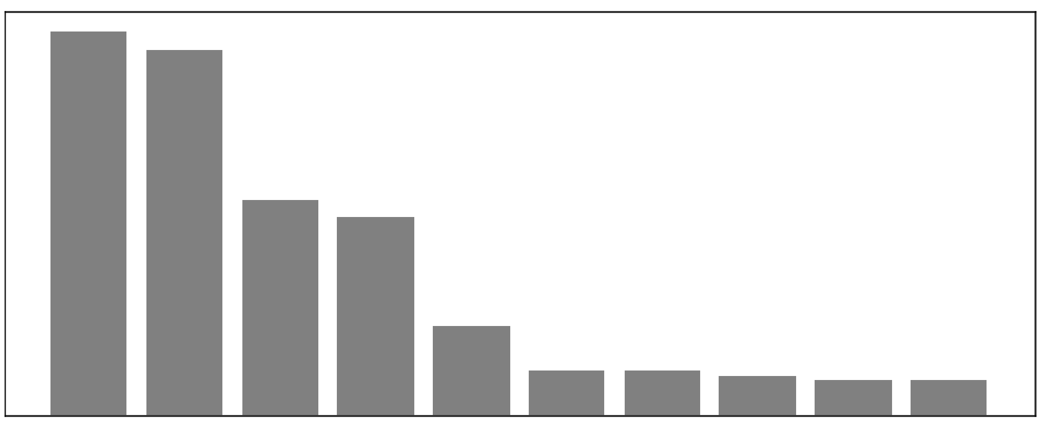
```
# Creating frequency distributions for each label to visually inspect
freq_dict = {}

for target in df.cyberbullying_type.unique():
    freq_dict[target] = FreqDist(df.loc[df.cyberbullying_type == targ

    top_10 = list(zip(*freq_dict[target].most_common(10)))
    tokens = top_10[0]
    counts = top_10[1]

    fig = plt.figure(figsize=(10,4))
    plt.bar(tokens,counts, color = 'grey')
    plt.title('Top tokens for '+target)
    plt.show()
```

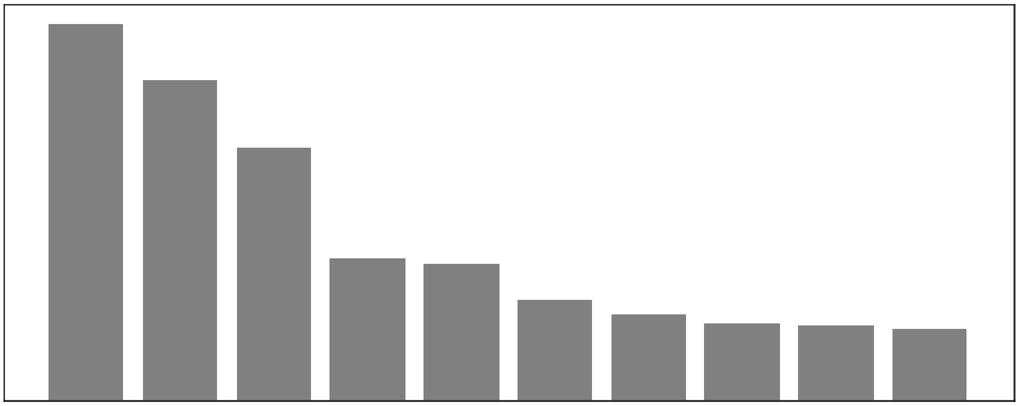












Observing Bigrams and Trigrams

In [11]:

```
# creating bigrams
def bigram(tweet):
    return [' '.join(word) for word in ngrams(tweet, 2)]

#creating trigrams
def _3gram(tweet):
    return [' '.join(word) for word in ngrams(tweet, 3)]

#adding to df
df['bigram'] = df.tokens.apply(bigram)
df['_3gram'] = df.tokens.apply(_3gram)
```

In [12]:

```
# looking at most common trigrams in dataset
FreqDist(df._3gram.explode()).most_common(10)
```

```
[('bulli high school', 1539),
 (nan, 1127),
 ('dumb ass nixxxx', 1104),
 ('fuck obama dumb', 979),
 ('obama dumb ass', 966),
 ('tayyoung_ fuck obama', 956),
 ('girl bulli high', 824),
 ('high school bulli', 653),
 ('rt tayyoung_ fuck', 456),
 ('girl high school', 415)]
```

Splitting Tweets and Labels into two dataframes

In [13]:
```
# df of features only
X = pd.DataFrame(df.tweet_text)

# df of labels
y = df.cyberbullying_type
```

Performing Train Test Split for Data Validation

In [14]:
```
# Split the data into Train and Test, so we can later validate our mc
# We must split before vectorizing, to simulate the real world where
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0
```

MODELING

We will first try classify cyberbullying using a multinomial bayes classifier.

In [15]:
```
# creating a function to evaluate models
def evaluate_multinomialnb(tfidf, model, X_train, X_test):

    # vectorizing data and fitting on training data
    X_train_v = tfidf.fit_transform(X_train.tweet_text)
    X_test_v = tfidf.transform(X_test.tweet_text)

    # computing mean accuracy
    train_acc = cross_val_score(model, X_train_v, y_train).mean()
    test_acc = cross_val_score(model, X_test_v, y_test).mean()

    # printing result
    print('Training Accuracy: {:.1%}'.format(train_acc))
    print('Testing Accuracy: {:.1%}'.format(test_acc))
```

Baseline Model

In [16]:

```
# initializing basic TFIDF vectorizer with 50 words with highest tfid
tfidf_1 = TfidfVectorizer(max_features=50)
model_1 = MultinomialNB()

evaluate_multinomialnb(tfidf_1, model_1, X_train, X_test)

print('''\nOur baseline model performs ok, if we were to randomly cla
has an accuracy of over 3x better. However this model will still pred
removing stopwords.''')
```

```
Training Accuracy: 56.1%
Testing Accuracy: 55.9%

Our baseline model performs ok, if we were to randomly classify cyberbullying we woul
d get an accuracy of 16.6%, whereas our model
has an accuracy of over 3x better. However this model will still predict correctly on
ly about half the time. Let's try to improve it by
removing stopwords.
```

Multinomial NB Model #2 - Removing Stopwords

In [17]:

```
# Instantiate the vectorizer with stopwords removed
tfidf_2 = TfidfVectorizer(
    max_features=50,
    stop_words=stopwords_list
)

evaluate_multinomialnb(tfidf_2, model_1, X_train, X_test)

print('''\n Our model has significantly improved once removing stopwo
words are considered different when they have the same underlying mea
```

```
Training Accuracy: 64.1%
Testing Accuracy: 66.1%

 Our model has significantly improved once removing stopwords. Let's continue to twea
k by stemming words, as we saw earlier that some
words are considered different when they have the same underlying meaning
```

Multinomial NB Model #3 - Removed Stopwords and Stemmed

In [18]:

```
# Instantiate the vectorizer with stemmed words
tfidf_3 = TfidfVectorizer(
    max_features = 50,
    stop_words = stopwords_list,
    token_pattern = r"(?u)\b\w+\b" #Changing tokens to include single
)
# calling function to evaluate
evaluate_multinomialnb(tfidf_3, model_1, X_train, X_test)

print('''\n Still improving slightly. lets add bigrams and increase m
```

```
Training Accuracy: 64.9%
Testing Accuracy: 66.5%

 Still improving slightly. lets add bigrams and increase max_features
```

Multinomial NB Model #4 - added Bigrams

In [19]:

```
# Instantiate the vectorizer with added Bigrams. also increased max_f
tfidf_4 = TfidfVectorizer(
    max_features = 80,
    stop_words = stopwords_list,
    token_pattern = r"(?u)\b\w+\b",
    ngram_range = (1, 2)
)
# calling function to evaluate
evaluate_multinomialnb(tfidf_4, model_1, X_train, X_test)

print('''\n Lets do some feature engineering to see if we can improve
```

```
Training Accuracy: 69.3%
Testing Accuracy: 69.7%

 Lets do some feature engineering to see if we can improve accuracy while taking orde
r of words into account
```

Multinomial NB Model #5 - feature engineering

In [20]:

```
# creating a column that notes if a tweet has a link or not
def has_link(x):
    if 'https' in x:
        return 1
    else:
        return 0

X_train['has_link'] = X_train.tweet_text.apply(has_link)
X_test['has_link'] = X_test.tweet_text.apply(has_link)
```

In [21]:

```
# creating a column that notes if a tweet is a reply or not
def is_reply(x):
    if '@' in x:
        return 1
    else:
        return 0

# applying new features to data
X_train['is_reply'] = X_train.tweet_text.apply(is_reply)
X_test['is_reply'] = X_test.tweet_text.apply(is_reply)

# resetting indices
X_train.reset_index(inplace=True)
X_test.reset_index(inplace=True)
```

In [22]:

```
# fitting and transforming X_train with tfidf used in previous model
X_train_v2 = tfidf_4.fit_transform(X_train.tweet_text)

# adding engineered features to X_train
X_train_2 = pd.DataFrame.sparse.from_spmatrix(X_train_v2, columns=tfi
X_train_2 = pd.concat([X_train_2, X_train[['has_link', 'is_reply']]],

# transforming X_test
X_test_v2 = tfidf_4.transform(X_test.tweet_text)

# adding engineered features to X_test
X_test_2 = pd.DataFrame.sparse.from_spmatrix(X_test_v2, columns=tfidf
X_test_2 = pd.concat([X_test_2, X_test[['has_link', 'is_reply']]], ax

# getting accuracy of X_train and X_test
train_acc = cross_val_score(model_1, X_train_2, y_train).mean()
test_acc = cross_val_score(model_1, X_test_2, y_test).mean()

#printing result
print('Training Accuracy: {:.1%}'.format(train_acc))
print('Testing Accuracy: {:.1%}'.format(test_acc))
print('''\nHere we have improved our model even more. Testing accurac
We can continue to add more features and start tuning, but let's firs
can get any promising results that way.''')
```

```
Training Accuracy: 72.5%
Testing Accuracy: 72.8%

Here we have improved our model even more. Testing accuracy indicates we are not over
fitting at all.
We can continue to add more features and start tuning, but let's first take a look at
recurrent neural networks and see if we
can get any promising results that way.
```

Recurrent Neural Networks

Bigrams and trigrams have clearly improved the model's ability to classify cyberbulling, however we are still losing knowledge with our bag of words approach, rather than having the ability to understand the complete order of the sentence. A recurrent neural network can help with this, and potentially give us a higher accuracy.

Recurrent Neural Networks take in the output of the first word, and use it as the input for the next run.

Reprocessing data to fit into neural network model

In [23]:

```
# tokenizing X_train_data
tweets = X_train.tweet_text.apply(tokenize)

# creating total vocabulary by using a set and comprehension. This wi
total_vocabulary = set(word.lower() for tweet in tweets for word in t
```

In [24]:

```
# encoding labels
y_train_d = pd.get_dummies(y_train).values
y_test_d = pd.get_dummies(y_test).values

# use keras to create a Tokenizer object
tokenizer = text.Tokenizer(num_words=20000)  # limiting number of wor

# giving each word a unique integer
tokenizer.fit_on_texts(list(X_train.tweet_text))

# creating a sequence of the created unique integers for each tweet f
tokenized_X_train = tokenizer.texts_to_sequences(X_train.tweet_text)

# Transforming X_test as well
tokenized_X_test  = tokenizer.texts_to_sequences(X_test.tweet_text)

# finally, padding each tweet in X_train so they are all the length o
X_train_pad = sequence.pad_sequences(tokenized_X_train, maxlen=140)

# padding test data
X_test_pad = sequence.pad_sequences(tokenized_X_test, maxlen=140)
```

In [25]:

```
# Running first Recurrent Neural Network Model
rnn_model_1 = Sequential() # Initializing sequential rnn_model_1
rnn_model_1.add(Embedding(len(total_vocabulary), 140)) # Embedding to
rnn_model_1.add(LSTM(25, return_sequences=True)) # Adding the Long Sh
rnn_model_1.add(GlobalMaxPool1D()) # Downsamples the input representa
rnn_model_1.add(Dense(50, activation='relu')) # Adding hidden layer w
rnn_model_1.add(Dense(50, activation='relu')) # Adding another hidden
rnn_model_1.add(Dense(6, activation='softmax'))  # 6 neurons because
```

In [26]:

```
rnn_model_1.compile(loss='categorical_crossentropy', # using this los
                    optimizer='adam', # an extension to stochastic gradient
                    metrics=['accuracy']) # using accuracy to evaluate our
rnn_model_1.summary() # printing summary
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 140)         7099680
_________________________________________________________________
lstm (LSTM)                  (None, None, 25)          16600
_________________________________________________________________
global_max_pooling1d (Global (None, 25)                0
_________________________________________________________________
dense (Dense)                (None, 50)                1300
_________________________________________________________________
dense_1 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 306
=================================================================
Total params: 7,120,436
Trainable params: 7,120,436
Non-trainable params: 0
_________________________________________________________________
```

In [27]:

```
rnn_model_1.fit(X_train_pad, y_train_d, epochs=5, batch_size=32, vali
```

```
Epoch 1/5
1006/1006 [==============================] - 80s 80ms/step - loss: 0.5957 - accuracy:
0.7416 - val_loss: 0.4504 - val_accuracy: 0.8155
Epoch 2/5
1006/1006 [==============================] - 80s 79ms/step - loss: 0.3436 - accuracy:
0.8589 - val_loss: 0.4307 - val_accuracy: 0.8306
Epoch 3/5
1006/1006 [==============================] - 81s 80ms/step - loss: 0.2523 - accuracy:
0.9017 - val_loss: 0.4847 - val_accuracy: 0.8214
Epoch 4/5
1006/1006 [==============================] - 80s 80ms/step - loss: 0.1909 - accuracy:
0.9265 - val_loss: 0.5216 - val_accuracy: 0.8202
Epoch 5/5
1006/1006 [==============================] - 81s 81ms/step - loss: 0.1487 - accuracy:
0.9390 - val_loss: 0.5671 - val_accuracy: 0.8211
```

```
<tensorflow.python.keras.callbacks.History at 0x16d9725e0>
```

Our first RNN model had a high accuracy of 94% on it's training data, but a significantly lower validation accuracy of 82%. this indicates we are indeed overfitting. Let's try Regularization through Ridge Regression (L2)

In [29]:

```
# Running next Recurrent Neural Network rnn_model_2
rnn_model_2 = Sequential()

rnn_model_2.add(Embedding(len(total_vocabulary), 140))
rnn_model_2.add(LSTM(25, kernel_regularizer=regularizers.l2(0.005), r
rnn_model_2.add(GlobalMaxPool1D())
rnn_model_2.add(Dense(50, kernel_regularizer=regularizers.l2(0.005),
rnn_model_2.add(Dense(50, kernel_regularizer=regularizers.l2(0.005),
rnn_model_2.add(Dense(6, activation='softmax'))
```

In [30]:

```
# compiling second model and printing summary
rnn_model_2.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
rnn_model_2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, None, 140)         7099680   
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 25)          16600     
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 25)                0         
_________________________________________________________________
dense_3 (Dense)              (None, 50)                1300      
_________________________________________________________________
dense_4 (Dense)              (None, 50)                2550      
_________________________________________________________________
dense_5 (Dense)              (None, 6)                 306       
=================================================================
Total params: 7,120,436
Trainable params: 7,120,436
Non-trainable params: 0
_________________________________________________________________
```

In [31]:
```
# running second model
history_rnn_2 = rnn_model_2.fit(X_train_pad, y_train_d, epochs=5, bat
```

```
Epoch 1/5
1006/1006 [==============================] - 84s 83ms/step - loss: 0.8929 - accuracy: 0.7006 - val_loss: 0.5974 - val_accuracy: 0.7758
Epoch 2/5
1006/1006 [==============================] - 83s 82ms/step - loss: 0.5202 - accuracy: 0.8089 - val_loss: 0.5450 - val_accuracy: 0.8105
Epoch 3/5
1006/1006 [==============================] - 103s 102ms/step - loss: 0.4293 - accuracy: 0.8576 - val_loss: 0.5899 - val_accuracy: 0.8046
Epoch 4/5
1006/1006 [==============================] - 91s 90ms/step - loss: 0.3705 - accuracy: 0.8860 - val_loss: 0.5793 - val_accuracy: 0.8068
Epoch 5/5
1006/1006 [==============================] - 84s 84ms/step - loss: 0.3261 - accuracy: 0.9002 - val_loss: 0.5942 - val_accuracy: 0.8035
```

Our second RNN model closed the gap between training and testing accuracies, but is still overfitting. We can adjust the lambda value for L2, but first let's experiment with adding dropout layers. this randomly ignores neurons in the network

In [33]:
```
# Running second Recurrent Neural Network rnn_model_3
rnn_model_3 = Sequential()

rnn_model_3.add(Embedding(len(total_vocabulary), 140))
rnn_model_3.add(LSTM(25, return_sequences=True))
rnn_model_3.add(GlobalMaxPool1D())
rnn_model_3.add(Dropout(0.5)) # randomly ignoring half of the neurons
rnn_model_3.add(Dense(50, activation='relu'))
rnn_model_3.add(Dropout(0.5))
rnn_model_3.add(Dense(50, activation='relu'))
rnn_model_3.add(Dropout(0.5))
rnn_model_3.add(Dense(6, activation='softmax'))
```

In [34]:

```
# compiling second model and printing summary
rnn_model_3.compile(loss='categorical_crossentropy',
                optimizer='adam',
                metrics=['accuracy'])
rnn_model_3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 140)         7099680
_________________________________________________________________
lstm_2 (LSTM)                (None, None, 25)          16600
_________________________________________________________________
global_max_pooling1d_2 (Glob (None, 25)                0
_________________________________________________________________
dropout (Dropout)            (None, 25)                0
_________________________________________________________________
dense_6 (Dense)              (None, 50)                1300
_________________________________________________________________
dropout_1 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_7 (Dense)              (None, 50)                2550
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_8 (Dense)              (None, 6)                 306
=================================================================
Total params: 7,120,436
Trainable params: 7,120,436
Non-trainable params: 0
_________________________________________________________________
```

In [103]:

```
# running second model
history_rnn_3 = rnn_model_3.fit(X_train_pad, y_train_d, epochs=4, bat
```

```
Epoch 1/4
1006/1006 [==============================] - 91s 91ms/step - loss: 0.3576 - accuracy:
0.8623 - val_loss: 0.5193 - val_accuracy: 0.8186
Epoch 2/4
1006/1006 [==============================] - 91s 90ms/step - loss: 0.3214 - accuracy:
0.8810 - val_loss: 0.5489 - val_accuracy: 0.8219
Epoch 3/4
1006/1006 [==============================] - 87s 87ms/step - loss: 0.2917 - accuracy:
0.8965 - val_loss: 0.6193 - val_accuracy: 0.8191
Epoch 4/4
1006/1006 [==============================] - 90s 90ms/step - loss: 0.2699 - accuracy:
0.9066 - val_loss: 0.6639 - val_accuracy: 0.8141
```

Our third RNN model does not overfit onto the training data, and has a testing accuracy of 80%. Let's try to increase the L2 value from .005 to .05 to see if we can improve our accuracy.

In [37]:

```
# Running next Recurrent Neural Network rnn_model_4
rnn_model_4 = Sequential()

rnn_model_4.add(Embedding(len(total_vocabulary), 140))
rnn_model_4.add(LSTM(25, kernel_regularizer=regularizers.l2(0.05), re
rnn_model_4.add(GlobalMaxPool1D())
rnn_model_4.add(Dense(50, kernel_regularizer=regularizers.l2(0.05), a
rnn_model_4.add(Dense(50, kernel_regularizer=regularizers.l2(0.05), a
rnn_model_4.add(Dense(6, activation='softmax'))
```

In [38]:

```
# compiling second model and printing summary
rnn_model_4.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
rnn_model_4.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 140)         7099680
_________________________________________________________________
lstm_3 (LSTM)                (None, None, 25)          16600
_________________________________________________________________
global_max_pooling1d_3 (Glob (None, 25)                0
_________________________________________________________________
dense_9 (Dense)              (None, 50)                1300
_________________________________________________________________
dense_10 (Dense)             (None, 50)                2550
_________________________________________________________________
dense_11 (Dense)             (None, 6)                 306
=================================================================
Total params: 7,120,436
Trainable params: 7,120,436
Non-trainable params: 0
_________________________________________________________________
```

In [39]:

```
# running second model
history_rnn_4 = rnn_model_4.fit(X_train_pad, y_train_d, epochs=5, bat
```

```
Epoch 1/5
1006/1006 [==============================] - 95s 94ms/step - loss: 1.8610 - accuracy:
0.5345 - val_loss: 1.0090 - val_accuracy: 0.6142
Epoch 2/5
1006/1006 [==============================] - 88s 87ms/step - loss: 0.9010 - accuracy:
0.6397 - val_loss: 0.8826 - val_accuracy: 0.6402
Epoch 3/5
1006/1006 [==============================] - 79s 79ms/step - loss: 0.8123 - accuracy:
0.6841 - val_loss: 0.8345 - val_accuracy: 0.6763
Epoch 4/5
1006/1006 [==============================] - 79s 79ms/step - loss: 0.7429 - accuracy:
0.7458 - val_loss: 0.8469 - val_accuracy: 0.7590
Epoch 5/5
1006/1006 [==============================] - 80s 79ms/step - loss: 0.6412 - accuracy:
0.8211 - val_loss: 0.6878 - val_accuracy: 0.7696
```

Once we increased the coefficient our model did not overfit as much, however our previous model with dropout laters still outperforms this model

Model Evaluation

In [104]:

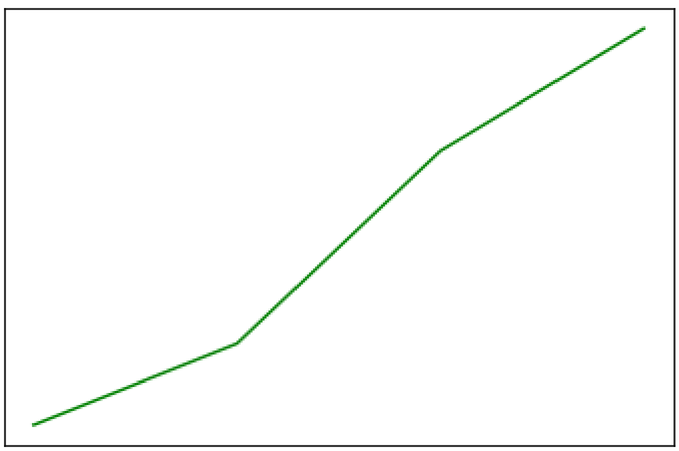
```
# Printing Results from RNN Models:
for idx, model in enumerate([rnn_model_1, rnn_model_2, rnn_model_3, r
    train_loss, train_acc = model.evaluate(X_train_pad, y_train_d)
    test_loss, test_acc = model.evaluate(X_test_pad, y_test_d)

    print('''
    Model: rnn_model_{}
    Train Accuracy = {:.1%}
    Test Accuracy = {:.1%}
    ------
    '''.format(idx+1, train_acc,test_acc))
```

```
1118/1118 [==============================] - 11s 10ms/step - loss: 0.1542 - accuracy:
0.9453
373/373 [==============================] - 4s 10ms/step - loss: 0.5106 - accuracy: 0.
8284

    Model: rnn_model_1
    Train Accuracy = 94.5%
    Test Accuracy = 82.8%
    ------

1118/1118 [==============================] - 11s 10ms/step - loss: 0.3189 - accuracy:
0.9122
373/373 [==============================] - 4s 10ms/step - loss: 0.5562 - accuracy: 0.
8054

    Model: rnn_model_2
    Train Accuracy = 91.2%
    Test Accuracy = 80.5%
    ------

1118/1118 [==============================] - 11s 10ms/step - loss: 0.2219 - accuracy:
0.9266
373/373 [==============================] - 4s 10ms/step - loss: 0.6199 - accuracy: 0.
8183

    Model: rnn_model_3
    Train Accuracy = 92.7%
    Test Accuracy = 81.8%
    ------

1118/1118 [==============================] - 11s 10ms/step - loss: 0.5746 - accuracy:
0.8402
373/373 [==============================] - 4s 10ms/step - loss: 0.6988 - accuracy: 0.
7640

    Model: rnn_model_4
    Train Accuracy = 84.0%
    Test Accuracy = 76.4%
    ------
```

After comparing the above models, RNN_model_3 is the best choice.

In [105]:

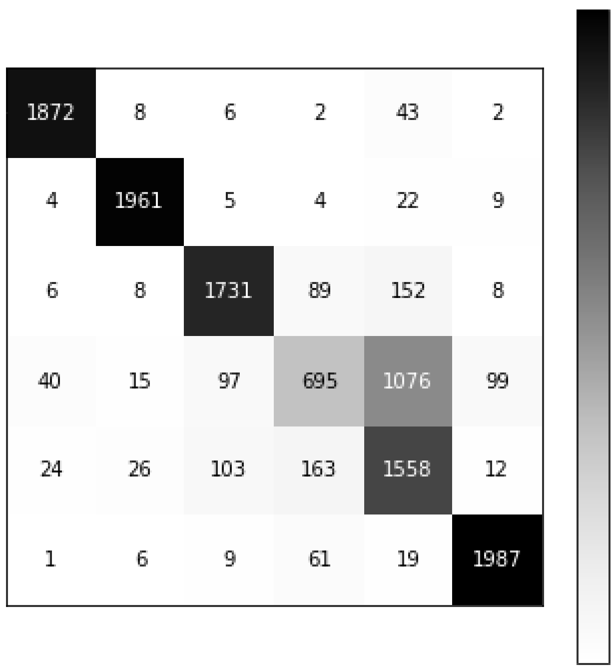
```
# Plot the loss vs the number of epoch
history_dict = history_rnn_3.history
loss_values = history_dict['val_loss']

epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, 'g', label='Training loss')

plt.title('Testing loss for RNN Model 5')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



In [54]:

```python
# Now that we have our final trained model, we will use the X_test tc
y_pred = np.argmax(rnn_model_3.predict(X_test_pad), axis = -1)

# need to rearrange y to be in one column for confusion matrix
y_test_labels = np.argmax(y_test_d,   axis=1)
y_train_labels = np.argmax(y_train_d, axis=1)

# creating a confusion matrix to observe accuracy between cyberbullyi
cm = confusion_matrix(y_test_labels, y_pred)

# displaying final cm
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['A
fig, ax = plt.subplots(figsize=(6,6))
disp.plot(cmap=plt.cm.Greys, ax = ax)
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x16debb730>
```



In [79]:

```
# evaluating confusion matrix
print('true non cyberbullying was correctly predicted {:.1%} of the t
print('"other" cyberbullying was incorrectly predicted {:.1%} of the
```

```
true non cyberbullying was correctly predicted 34.4% of the time
"other" cyberbullying was incorrectly predicted 54.2% of the time
```

FINAL MODEL EVALUATION:

The final model chosen is an RNN Model with 3 dropout layers to prevent overfitting. Testing Accuracy was 82.2% and Training Accuracy was 87.6%. This model had the highest testing accuracy, and minimal overfitting.

Since it is a recurrent model, the order of words are incorporated into the model. It also uses Long Term Short Memory cells which teaches the model which words are important and which ones we can forget. This will help our algorithm as we incorporate more and more data.

However, there are ways to improve this model. as seen in the confusion matrix, there is incorrect labeling between non-cyberbullying and other. Non-cyberbullying was only predicted 35% of the time. This would lead to many flagged tweets, therefore drasticly affecting the purpose of browsing twitter.

Conclusions and Next Steps

In the end, our Recurrent Neural Network Model outperformed our Multinomial Naive Bayes Model in terms of accuracy on both Training and Testing data. Our first Sequential model had an accuracy of 95% on it's training data, but was severely overfitting as we saw with our low Testing data score. With each iteration we worked towards reducing overfitting, through the dropout method and L2 regularization. In the end, we settled on our final model with a Training Accuracy of 87.6% and Testing Accuracy of 82.2%.

Next steps to consider when working on this project would be to create a pipeline to find the optimal testing accuracy, as it could be higher. I would also look into the low accuracy score between "No Cyberbullying" and "Other Cyberbullying", as our confusion matrix shows that the model could not identify differences in these tweets as well. This would involve potentially gathering more data, or looking more into the accuracy of labeling for those two categories. I would also incorporate data into this model that would have the ability to flag fake news, as these types of tweets could increase with less moderation on twitter, and are dangerous to our society.

In [ ]:

```
```