Tweet Sentiment Classification

Module 4 Project - Kai Graham

Overview of Process (CRISP-DM)

I will be following the Cross-Industry Standard Process for Data Mining to build a classifier that will determine the sentiment of tweets. The CRISP-DM process includes the following key steps:

1. Business Understanding
2. Data Understanding
3. Data Preparation
4. Modeling
5. Evaluation

1. Business Understanding

The overall goal of this process is to create a classifier that can label tweets based on their sentiment (positive, negative, or neutral). This classifier will be created in the context of tracking public sentiment surrounding various product releases / events. Stakeholders for this project are likely product managers / investor and public relations professionals who are invested in how the public is feeling towards newly released products, and various other business proceedings. In conjunction with other tools, companies could use this classifier to create a sentiment score based on a certain number of recent tweets, and track that over time to monitor changes in sentiment over time. Additionally, companies could pull down a batch of tweets at certain times, filtered for various products or topics. If a broad enough sample is used, it could likely help provide good insight into whether consumers are feeling neutral, positive, or negative towards recent releases.

According to [https://www.internetlivestats.com/twitter-statistics/ (https://www.internetlivestats.com/twitter-statistics/)](https://www.internetlivestats.com/twitter-statistics/), there are over 500 million tweets sent per day. Harnessing public sentiment from this amount of data would undoubtedly be helpful for tech companies looking to track how the public is feeling towards them. This classifier would likely be a valuable tool to complement public product reviews.

2. Data Understanding

```
The main dataset used throughout this data science process will be coming
from CrowdFlower via the following url:
`https://data.world/crowdflower/brands-and-product-emotions`.

The following summary of the dataset is provided on CrowdFlower:
```

Contributors evaluated tweets about multiple brands and products. The crowd was asked if the tweet expressed positive, negative, or no emotion towards a brand and/or product. If some emotion was expressed they were also asked to say which brand or product was the target of that emotion.

As the dataset contains labels classifying the tweet as positive, negative, or neutral, along with detailed tweet text, the dataset available is a good match for our business goals.

In [1]:
```
# import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import nltk
from nltk.corpus import stopwords
from nltk.collocations import *
import string
import re
from sklearn.metrics import accuracy_score, precision_score, recall_score,
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GridSearchCV, Randomi
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorize
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.svm import LinearSVC
from imblearn.over_sampling import SMOTE
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Input, Dense, LSTM, Embedding, Dropout, Activation
from keras.models import Sequential
from keras import initializers, regularizers, constraints, optimizers, laye
import tensorflow
```

```
Using TensorFlow backend.
```

In [2]:
```
# failure to specify 'latin1' encoding results in errors
# error_df = pd.read_csv('judge-1377884607_tweet_product_company.csv')
# error_df.head()
```

In [3]:
```
# load dataset
raw_df = pd.read_csv('judge-1377884607_tweet_product_company.csv', encoding
```

In [4]:
```
# print first rows of dataset
raw_df.head()
```

Out[4]:

	tweet_text	emotion_in_tweet_is_directed_at	is_there_an_emotion_directed_at_a_brand_or_product
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iPhone	Negative emotion
1	@jessedee Know about @fludapp ? Awesome iPad/i...	iPad or iPhone App	Positive emotion
2	@swonderlin Can not wait for #iPad 2 also. The...	iPad	Positive emotion
3	@sxsw I hope this year's festival isn't as cra...	iPad or iPhone App	Negative emotion
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	Google	Positive emotion

In [5]:
```
# show info of df
raw_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9093 entries, 0 to 9092
Data columns (total 3 columns):
 #   Column                                              Non-Null Count
Dtype
---  ------                                              --------------
-----
 0   tweet_text                                          9092 non-null
object
 1   emotion_in_tweet_is_directed_at                     3291 non-null
object
 2   is_there_an_emotion_directed_at_a_brand_or_product  9093 non-null
object
dtypes: object(3)
memory usage: 213.2+ KB
```

Looking at the above outputs, we can see that there are a total of 9,093 entries in our dataset, with a total of three columns. Raw text from tweets is held in the `tweet_text` column; sentiment is held in the `is_there_an_emotion_directed_at_a_brand_or_product` ; and the item of emotion direction is held in the `emotion_in_tweet_is_directed_at` column.

From first glance, we can likely drop the `emotion_in_tweet_is_directed_at` column as we are more interested in whether sentiment in a given tweet is positive, neutral, or negative based on the text. Main predictors we will use is processed features derived from the `tweet_text` column.

Our target variable, which can also be though of as our class labels are held in the `is_there_an_emotion_directed_at_a_brand_or_product` column.

In [6]:
```
# display value counts
display(raw_df['emotion_in_tweet_is_directed_at'].value_counts())
display(raw_df['is_there_an_emotion_directed_at_a_brand_or_product'].value_
```

```
iPad                               946
Apple                              661
iPad or iPhone App                 470
Google                             430
iPhone                             297
Other Google product or service    293
Android App                         81
Android                             78
Other Apple product or service      35
Name: emotion_in_tweet_is_directed_at, dtype: int64
```

```
No emotion toward brand or product    5389
Positive emotion                      2978
Negative emotion                       570
I can't tell                           156
Name: is_there_an_emotion_directed_at_a_brand_or_product, dtype: int64
```

```
Unsurprising given the origin of our dataset, the products identfied are
either Apple or Google products.  Looking at sentiment, the majority of
entries seem to fall under a neutral sentiment ('No emotion toward brand
or product'), with the next largest group being tagged as 'Positive
emotion'.  There is some clear class imbalance present with only 570
entries belonging to the 'Negative emotion' class. The lack of negtive
sentiment labels is likely a weakness of this dataset.
```

In [7]:
```
# rename columns so easier to work with
df = raw_df.copy()
df.columns = ['text', 'product_brand', 'sentiment']
df.head()
```

Out[7]:

	text	product_brand	sentiment
0	.@wesley83 I have a 3G iPhone. After 3 hrs twe...	iPhone	Negative emotion
1	@jessedee Know about @fludapp ? Awesome iPad/i...	iPad or iPhone App	Positive emotion
2	@swonderlin Can not wait for #iPad 2 also. The...	iPad	Positive emotion
3	@sxsw I hope this year's festival isn't as cra...	iPad or iPhone App	Negative emotion
4	@sxtxstate great stuff on Fri #SXSW: Marissa M...	Google	Positive emotion

In [8]:
```
# explore potential missing values
df.isna().sum()
```

Out[8]:
```
text                 1
product_brand     5802
sentiment            0
dtype: int64
```

We see that there is only one missing value in the text column, 0 in the sentiment column, and a large number (5802) in the product_brand column. Given we are planning to work the majority of the time with the text and sentiment columns, this will not likely pose a large issue.

In [9]:
```
# display missing value in the text column
df.loc[df['text'].isna()]
```

Out[9]:

	text	product_brand	sentiment
6	NaN	NaN	No emotion toward brand or product

In [10]:
```
# display missing values in the product_brand column
df.loc[df['product_brand'].isna()]
```

Out[10]:

	text	product_brand	sentiment
5	@teachntech00 New iPad Apps For #SpeechTherapy...	NaN	No emotion toward brand or product
6	NaN	NaN	No emotion toward brand or product
16	Holler Gram for iPad on the iTunes App Store -...	NaN	No emotion toward brand or product
32	Attn: All #SXSW frineds, @mention Register fo...	NaN	No emotion toward brand or product
33	Anyone at #sxsw want to sell their old iPad?	NaN	No emotion toward brand or product
...	...	...	...
9087	@mention Yup, but I don't have a third app yet...	NaN	No emotion toward brand or product
9089	Wave, buzz... RT @mention We interrupt your re...	NaN	No emotion toward brand or product
9090	Google's Zeiger, a physician never reported po...	NaN	No emotion toward brand or product
9091	Some Verizon iPhone customers complained their...	NaN	No emotion toward brand or product
9092	Ï¡ Ïà ü_ Ê Î Ò £ Á ââ _ £ â_ ÛâRT @...	NaN	No emotion toward brand or product

5802 rows × 3 columns

In [11]:
```
# display sentiment breakdowns of missing product_brand entries
df.loc[df['product_brand'].isna()]['sentiment'].value_counts()
```

Out[11]:
```
No emotion toward brand or product    5298
Positive emotion                       306
I can't tell                           147
Negative emotion                        51
Name: sentiment, dtype: int64
```

We see that the majority of missing product_brand values are also labeled as no emotion twoard brand or product, which makes sense as a lot of the neutral-labeled tweets may not be directed at a specific brand or product, and therefore would be missing a product_brand tagging. Additionally, this column will not be used in our process of tweet classification.

Drop unnecessary columns and handle missing value for additional EDA.

In [12]:
```
# drop product_brand column
clean_df = df.drop(['product_brand'], axis=1)

# handle missing values
clean_df = clean_df.dropna(subset=['text'])
clean_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9092 entries, 0 to 9092
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   text       9092 non-null   object
 1   sentiment  9092 non-null   object
dtypes: object(2)
memory usage: 213.1+ KB
```

In [13]:
```
# further examine tweets labeled as "I can't tell"
for i in range(10):
    display(clean_df.loc[clean_df['sentiment'] == "I can't tell"].iloc[i][0
```

```
'Thanks to @mention for publishing the news of @mention new medical Apps
at the #sxswi conf. blog {link} #sxsw #sxswh'

'\x89ÛÏ@mention &quot;Apple has opened a pop-up store in Austin so the ne
rds in town for #SXSW can get their new iPads. {link} #wow'

'Just what America needs. RT @mention Google to Launch Major New Social N
etwork Called Circles, Possibly Today {link} #sxsw'

'The queue at the Apple Store in Austin is FOUR blocks long. Crazy stuff!
#sxsw'

"Hope it's better than wave RT @mention Buzz is: Google's previewing a so
cial networking platform at #SXSW: {link}"

'SYD #SXSW crew your iPhone extra juice pods have been procured.'

'Why Barry Diller thinks iPad only content is nuts @mention #SXSW {link}'

'Gave into extreme temptation at #SXSW and bought an iPad 2... #impulse'

'Catch 22\x89Û_ I mean iPad 2 at #SXSW : {link}'

'Forgot my iPhone for #sxsw. Android only. Knife to a gun fight'
```

Looking at a sample of the tweets labeled as "I can't tell", there is no clear class label that each should belong to. Given this, and the small number of tweets with this class distinction, they will be removed from the dataset.

In [14]:
```
# separate dataset into tweets and class_labels for additional EDA
tweets = clean_df['text']
class_labels = clean_df['sentiment']
```

In [15]:
```
# tokenize tweets and print the total vocabulary size of our dataset
tokenized = list(map(nltk.word_tokenize, tweets.dropna()))
raw_tweet_vocab = set()
for tweet in tokenized:
    raw_tweet_vocab.update(tweet)
print(len(raw_tweet_vocab))
```

```
13212
```

Looking at the text within the tweets, there is a total vocabulary size of just over 13,200.

In [16]:
```
# print average tweet size
mean_tweet_size = []
for tweet in tokenized:
    mean_tweet_size.append(len(tweet))
np.mean(mean_tweet_size)
```

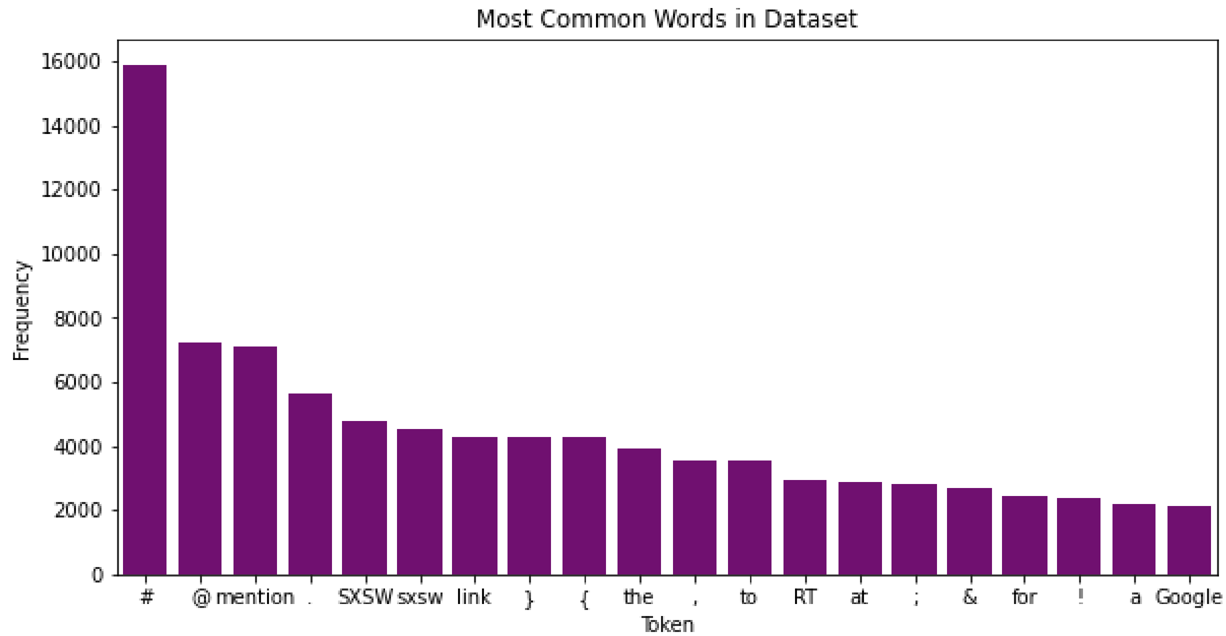
Out[16]:
```
24.414980202375716
```

The average tweet size within the dataset is just over 24 words.

In [17]:
```
# display frequency distribution of raw dataset
tweets_concat = []
for tweet in tokenized:
    tweets_concat += tweet

# display the 15 most common words
unprocessed_freq_dist = nltk.FreqDist(tweets_concat)
unprocessed_freq_dist.most_common(25)
```

Out[17]:
```
[('#', 15875),
 ('@', 7194),
 ('mention', 7123),
 ('.', 5601),
 ('SXSW', 4787),
 ('sxsw', 4523),
 ('link', 4311),
 ('}', 4298),
 ('{', 4296),
 ('the', 3928),
 (',', 3533),
 ('to', 3521),
 ('RT', 2947),
 ('at', 2859),
 (';', 2800),
 ('&', 2707),
 ('for', 2440),
 ('!', 2398),
 ('a', 2174),
 ('Google', 2136),
 ('iPad', 2129),
 (':', 2075),
 ('Apple', 1882),
 ('in', 1833),
 ('quot', 1696)]
```

In [18]:
```
# visualize frequency distribution
top_20 = pd.DataFrame(unprocessed_freq_dist.most_common(20), columns=['toke
plt.figure(figsize=(10, 5))
sns.barplot(x=top_20['token'], y=top_20['freq'], color='purple')
plt.xlabel('Token')
plt.ylabel('Frequency')
plt.title('Most Common Words in Dataset')
plt.show()
```



From first glance, we can see a number of the top appear words / tokens are stopwords or punctuation. Not surprising given our dataset, we also see the most common tokens are related to twitter tweet structure, with `#` and `@` along with others standing out. For additional EDA processed, we will try removing stopwords to see if additional information can be extracted from the data.

In [19]:
```
def initial_tweet_process(tweet, stopwords_list):
    """
    Function to intially process a tweet to assist in EDA / data understand
    Input: tweet of type string, stopwords_list of words to remove
    Returns: tokenized tweet, converted to lowercase, with all stopwords rem
    """
    # tokenize
    tokens = nltk.word_tokenize(tweet)

    # remove stopwords and lowercase
    stopwords_removed = [token.lower() for token in tokens if token.lower()

    # return processed tweet
    return stopwords_removed
```

In [20]:
```
def concat_tweets(tweets):
    """
    Function to concatenate a list of tweets into one piece of text.
    Input: tweets (list of tweets)
    Returns: concatenated tweet string
    """
    tweets_concat = []
    for tweet in tweets:
        tweets_concat += tweet
    return tweets_concat
```

In [21]:
```
def process_concat(raw_text, stopwords_list):
    """
    Function to process and return concatenated tweets.  Takes raw text, an
    Returns: concatenated tweets
    """
    processed_text = raw_text.apply(lambda x: initial_tweet_process(x, stop
    return concat_tweets(list(processed_text))
```

In [22]:
```
def print_normalized_word_freq(freq_dist, n=15):
    """
    Print a normalized frequency distribution from a given distribution. Re
    """
    total_word_count = sum(freq_dist.values())
    top = freq_dist.most_common(n)

    print('Word\t\t\tNormalized Frequency')
    for word in top:
        normalized_freq = word[1] / total_word_count
        print('{} \t\t\t {:.4}'.format(word[0], normalized_freq))

    return None
```

In [23]:
```
def print_bigrams(tweets_concat, n=15):
    """
    Function takes concatenated tweets and prints most common bigrams
    """
    bigram_measures = nltk.collocations.BigramAssocMeasures()
    finder = BigramCollocationFinder.from_words(tweets_concat)
    tweet_scored = finder.score_ngrams(bigram_measures.raw_freq)
    display(tweet_scored[:n])
    return tweet_scored
```

In [24]:
```
def display_pmi(tweets_concat, freq_filter=10, n=15):
    """
    Function that takes concatenated tweets and a freq_filter number. Displ
    """
    bigram_measures = nltk.collocations.BigramAssocMeasures()
    tweet_pmi_finder = BigramCollocationFinder.from_words(tweets_concat)
    tweet_pmi_finder.apply_freq_filter(freq_filter)
    tweet_pmi_scored = tweet_pmi_finder.score_ngrams(bigram_measures.pmi)
    display(tweet_pmi_scored[:n])
    return tweet_pmi_scored
```

In [25]:
```
# set up initial stopwords list
stopwords_list = stopwords.words('english') + list(string.punctuation)
stopwords_list += ["''", '""', '...', '``']
stopwords_list += ['mention', 'sxsw', 'link', 'rt', 'quot', 'google', 'appl
```

In [26]:
```
# separate dataset based on class label
neutral_tweets = clean_df.loc[clean_df['sentiment'] == 'No emotion toward b
positive_tweets = clean_df.loc[clean_df['sentiment'] == 'Positive emotion']
negative_tweets = clean_df.loc[clean_df['sentiment'] == 'Negative emotion']
ambig_tweets = clean_df.loc[clean_df['sentiment'] == "I can't tell"]
all_tweets = clean_df.copy()
```

In [27]:
```
# process and concat datasets
concat_neutral = process_concat(neutral_tweets['text'], stopwords_list)
concat_positive = process_concat(positive_tweets['text'], stopwords_list)
concat_negative = process_concat(negative_tweets['text'], stopwords_list)
concat_ambig = process_concat(ambig_tweets['text'], stopwords_list)
concat_all = process_concat(all_tweets['text'], stopwords_list)
```

In [28]:
```
# produce freq dists
freqdist_neutral = nltk.FreqDist(concat_neutral)
freqdist_positive = nltk.FreqDist(concat_positive)
freqdist_negative = nltk.FreqDist(concat_negative)
freqdist_ambig = nltk.FreqDist(concat_ambig)
freqdist_all = nltk.FreqDist(concat_all)
```

In [29]:
```
# display top neutral words
print('Top Neutral Words')
neutral_top_15 = freqdist_neutral.most_common(15)
neutral_top_15
```

```
Top Neutral Words
```

Out[29]:
```
[('ipad', 1212),
 ('store', 867),
 ('iphone', 815),
 ('new', 678),
 ("'s", 648),
 ('austin', 630),
 ('amp', 601),
 ('2', 550),
 ('circles', 490),
 ('social', 481),
 ('launch', 465),
 ('today', 441),
 ('app', 355),
 ('network', 355),
 ('android', 350)]
```

In [30]:
```
# display top positive words
print('Top Positive Words')
positive_top_15 = freqdist_positive.most_common(15)
positive_top_15
```

```
Top Positive Words
```

Out[30]:
```
[('ipad', 1003),
 ('store', 545),
 ('iphone', 523),
 ("'s", 493),
 ('2', 490),
 ('app', 396),
 ('new', 360),
 ('austin', 294),
 ('amp', 211),
 ('ipad2', 209),
 ('android', 198),
 ('launch', 160),
 ('get', 157),
 ("n't", 152),
 ('pop-up', 151)]
```

In [31]:
```
# display negative words
print('Top Negative Words')
negative_top_15 = freqdist_negative.most_common(15)
negative_top_15
```

```
Top Negative Words
```

Out[31]:
```
[('ipad', 188),
 ('iphone', 162),
 ("n't", 87),
 ("'s", 77),
 ('2', 64),
 ('app', 60),
 ('store', 46),
 ('new', 43),
 ('like', 39),
 ('circles', 34),
 ('social', 31),
 ('apps', 30),
 ('people', 29),
 ('design', 28),
 ('need', 25)]
```

In [32]:
```
print('Top Ambiguous Words')
ambig_top_15 = freqdist_ambig.most_common(15)
ambig_top_15
```

```
Top Ambiguous Words
```

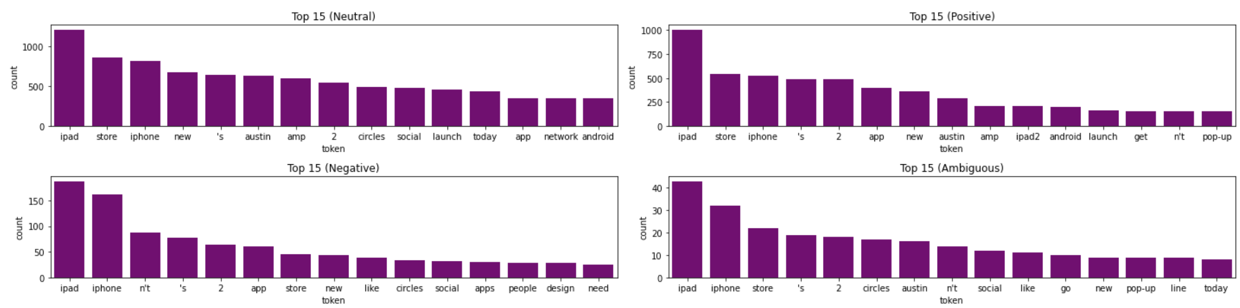
Out[32]:
```
[('ipad', 43),
 ('iphone', 32),
 ('store', 22),
 ("'s", 19),
 ('2', 18),
 ('circles', 17),
 ('austin', 16),
 ("n't", 14),
 ('social', 12),
 ('like', 11),
 ('go', 10),
 ('new', 9),
 ('pop-up', 9),
 ('line', 9),
 ('today', 8)]
```

In [33]:
```
# display top all
print('Top Words from All Tweets')
all_top_15 = freqdist_all.most_common(15)
all_top_15
```

```
Top Words from All Tweets
```

Out[33]:
```
[('ipad', 2446),
 ('iphone', 1532),
 ('store', 1480),
 ("'s", 1237),
 ('2', 1122),
 ('new', 1090),
 ('austin', 964),
 ('amp', 836),
 ('app', 817),
 ('circles', 658),
 ('launch', 653),
 ('social', 648),
 ('today', 580),
 ('android', 577),
 ("n't", 482)]
```

In [34]:
```
# visualize with subplots
fig, axes = plt.subplots(2, 2, figsize=(20, 5))
freq_dists = [neutral_top_15, positive_top_15, negative_top_15, ambig_top_1
labels = ['Top 15 (Neutral)', 'Top 15 (Positive)', 'Top 15 (Negative)', 'To
for idx, ax in enumerate(axes.flat):
    sns.barplot(data=pd.DataFrame(freq_dists[idx], columns=['token', 'count
                x='token',
                y='count',
                ax=ax,
                color='purple')
    ax.set_title(labels[idx])
plt.tight_layout()
```



Comparing the top words, we see again that the following words appear frequently in all class labels, and are therefore not as helpful in classification. Update stopwords list and reprint frequency distributions.

In [35]:
```
# create additional stopwords
additional_stopwords = ['ipad', 'iphone', 'android', 'store']
```

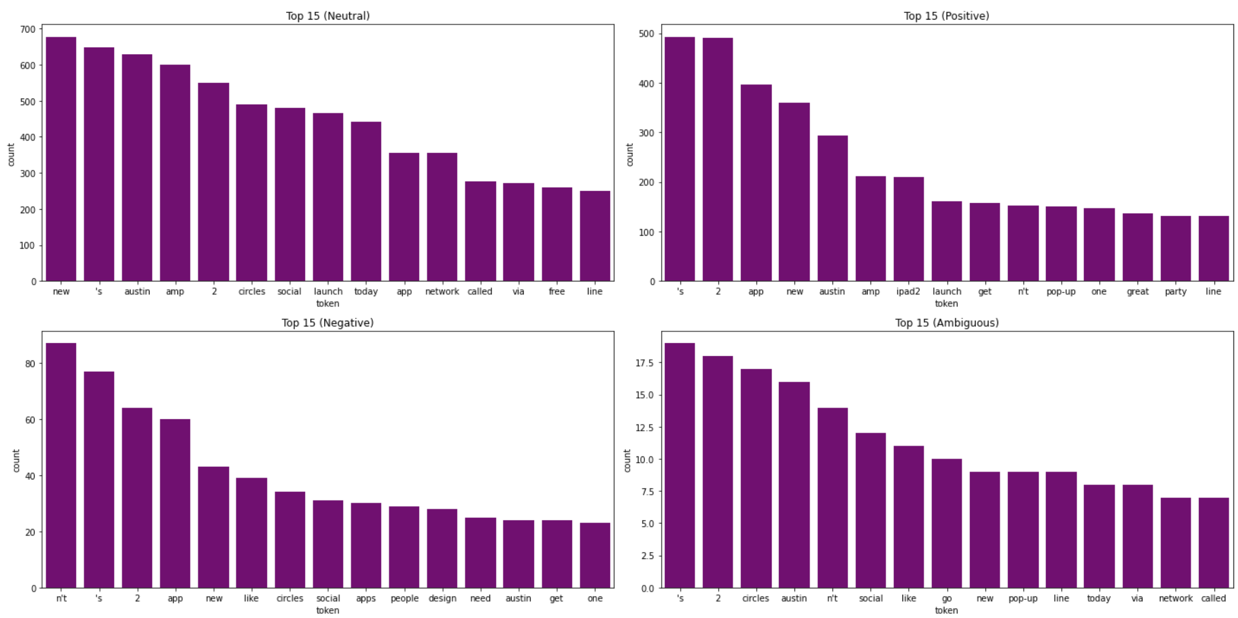
In [36]:
```
new_stopwords = stopwords_list + additional_stopwords
```

In [37]:
```
# process and concat datasets
concat_neutral = process_concat(neutral_tweets['text'], new_stopwords)
concat_positive = process_concat(positive_tweets['text'], new_stopwords)
concat_negative = process_concat(negative_tweets['text'], new_stopwords)
concat_ambig = process_concat(ambig_tweets['text'], new_stopwords)
concat_all = process_concat(all_tweets['text'], new_stopwords)
```

In [38]:
```
# produce frequency distributions
freqdist_neutral = nltk.FreqDist(concat_neutral)
freqdist_positive = nltk.FreqDist(concat_positive)
freqdist_negative = nltk.FreqDist(concat_negative)
freqdist_ambig = nltk.FreqDist(concat_ambig)
freqdist_all = nltk.FreqDist(concat_all)
```

In [132]:

```
# revisualize after removing additional stopwords
fig, axes = plt.subplots(2, 2, figsize=(20, 10))
freq_dists = [freqdist_neutral.most_common(15), freqdist_positive.most_comm
              freqdist_negative.most_common(15), freqdist_ambig.most_common
labels = ['Top 15 (Neutral)', 'Top 15 (Positive)', 'Top 15 (Negative)', 'To
for idx, ax in enumerate(axes.flat):
    sns.barplot(data=pd.DataFrame(freq_dists[idx], columns=['token', 'count
                x='token',
                y='count',
                ax=ax,
                color='purple')
    ax.set_title(labels[idx])
plt.tight_layout()
```



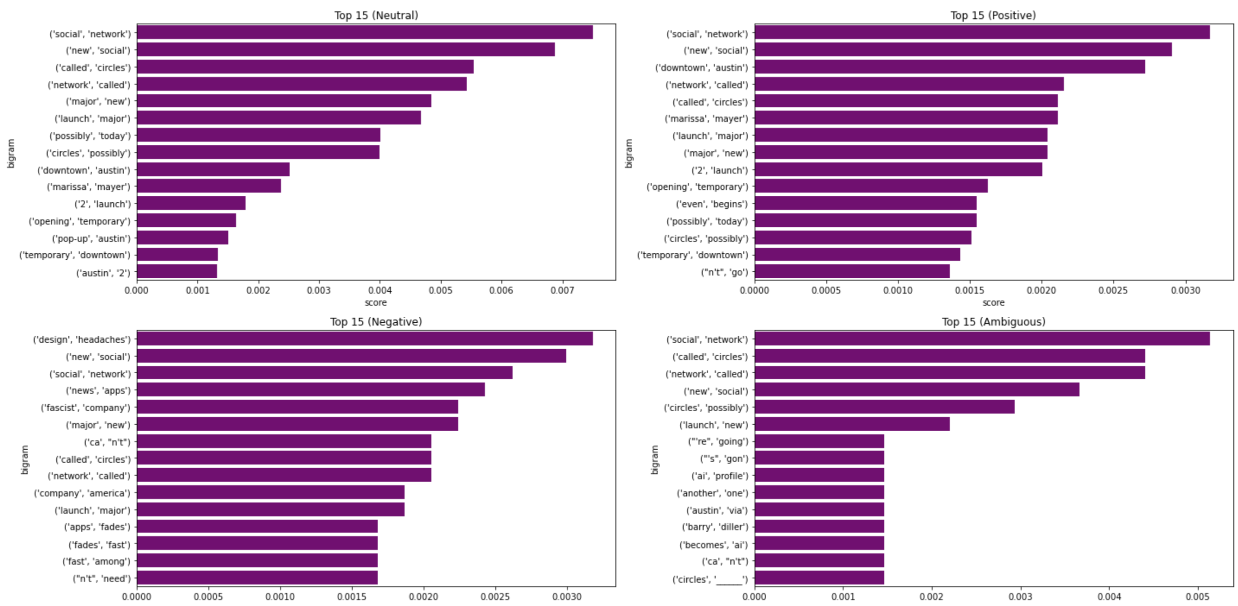
With additional stopwords removed and some processing applied, we can start to see more of a pattern appearing. Looking at the positive words, tokens that now stand out include launch, great, and party; whereas the following tokens stand out for the more common words in negative tweets: design, need

In [40]:
```
# print bigrams
print('Neutral Bigrams:')
neutral_bigrams = print_bigrams(concat_neutral)

print('Positive Bigrams')
positive_bigrams = print_bigrams(concat_positive)

print('Negative Bigrams')
negative_bigrams = print_bigrams(concat_negative)

print('Ambiguous Bigrams')
ambig_bigrams = print_bigrams(concat_ambig)
```

```
Neutral Bigrams:

[(('social', 'network'), 0.007498383968972204 5),
 (('new', 'social'), 0.006873518638224520 4),
 (('called', 'circles'), 0.005537599655246714),
 (('network', 'called'), 0.005429864253393665),
 (('major', 'new'), 0.004848093083387201),
 (('launch', 'major'), 0.004675716440422323),
 (('possibly', 'today'), 0.004007756948933419),
 (('circles', 'possibly'), 0.003986209868562809 5),
 (('downtown', 'austin'), 0.002521008403361344 7),
 (('marissa', 'mayer'), 0.002370178840767076),
 (('2', 'launch'), 0.001788407670760611 9),
 (('opening', 'temporary'), 0.001637578108166343 5),
 (('pop-up', 'austin'), 0.001508295625942684 8),
 (('temporary', 'downtown'), 0.001335918982977806 5),
 (('austin', '2'), 0.001314371902607196 8)]

Positive Bigrams

[(('social', 'network'), 0.003172684695573349 6),
 (('new', 'social'), 0.002908294304275570 5),
 (('downtown', 'austin'), 0.002719444024777156 6),
 (('network', 'called'), 0.002152893186281915 6),
 (('called', 'circles'), 0.002115123130382233),
 (('marissa', 'mayer'), 0.002115123130382233),
 (('launch', 'major'), 0.002039583018582867 6),
 (('major', 'new'), 0.002039583018582867 6),
 (('2', 'launch'), 0.002001812962683185),
 (('opening', 'temporary'), 0.001624112403686357 4),
 (('even', 'begins'), 0.001548572291886992),
 (('possibly', 'today'), 0.001548572291886992),
 (('circles', 'possibly'), 0.001510802235987309 2),
 (('temporary', 'downtown'), 0.001435262124187943 9),
 (("n't", 'go'), 0.001359722012388578 3)]

Negative Bigrams

[(('design', 'headaches'), 0.003181137724550898),
 (('new', 'social'), 0.002994011976047904 4),
 (('social', 'network'), 0.002619760479041916),
 (('news', 'apps'), 0.002432634730538922 2),
 (('fascist', 'company'), 0.002245508982035928),
 (('major', 'new'), 0.002245508982035928),
```

```
 (('ca', "n't"), 0.0020583832335329343),
 (('called', 'circles'), 0.0020583832335329343),
 (('network', 'called'), 0.0020583832335329343),
 (('company', 'america'), 0.0018712574850299401),
 (('launch', 'major'), 0.0018712574850299401),
 (('apps', 'fades'), 0.0016841317365269462),
 (('fades', 'fast'), 0.0016841317365269462),
 (('fast', 'among'), 0.0016841317365269462),
 (("n't", 'need'), 0.0016841317365269462)]

Ambiguous Bigrams

[(('social', 'network'), 0.005139500734214391),
 (('called', 'circles'), 0.004405286343612335),
 (('network', 'called'), 0.004405286343612335),
 (('new', 'social'), 0.003671071953010279),
 (('circles', 'possibly'), 0.0029368575624082223),
 (('launch', 'new'), 0.0022026431718061676),
 (("'re", 'going'), 0.0014684287812041115),
 (("'s", 'gon'), 0.0014684287812041115),
 (('ai', 'profile'), 0.0014684287812041115),
 (('another', 'one'), 0.0014684287812041115),
 (('austin', 'via'), 0.0014684287812041115),
 (('barry', 'diller'), 0.0014684287812041115),
 (('becomes', 'ai'), 0.0014684287812041115),
 (('ca', "n't"), 0.0014684287812041115),
 (('circles', '______'), 0.0014684287812041115)]
```

In [41]:
```
# pull out top n bigrams
top_n = 15
top_neutral_bigrams = pd.DataFrame(neutral_bigrams[:top_n], columns=['bigra
top_positive_bigrams = pd.DataFrame(positive_bigrams[:top_n], columns=['big
top_negative_bigrams = pd.DataFrame(negative_bigrams[:top_n], columns=['big
top_ambig_bigrams = pd.DataFrame(ambig_bigrams[:top_n], columns=['bigram',

# visualize
fig, axes = plt.subplots(2, 2, figsize=(20, 10))
bigrams = [top_neutral_bigrams, top_positive_bigrams, top_negative_bigrams,
labels = ['Top 15 (Neutral)', 'Top 15 (Positive)', 'Top 15 (Negative)', 'To
for idx, ax in enumerate(axes.flat):
    sns.barplot(data=bigrams[idx],
                y='bigram',
                x='score',
                ax=ax,
                color='purple')
    ax.set_title(labels[idx])
plt.tight_layout()
```

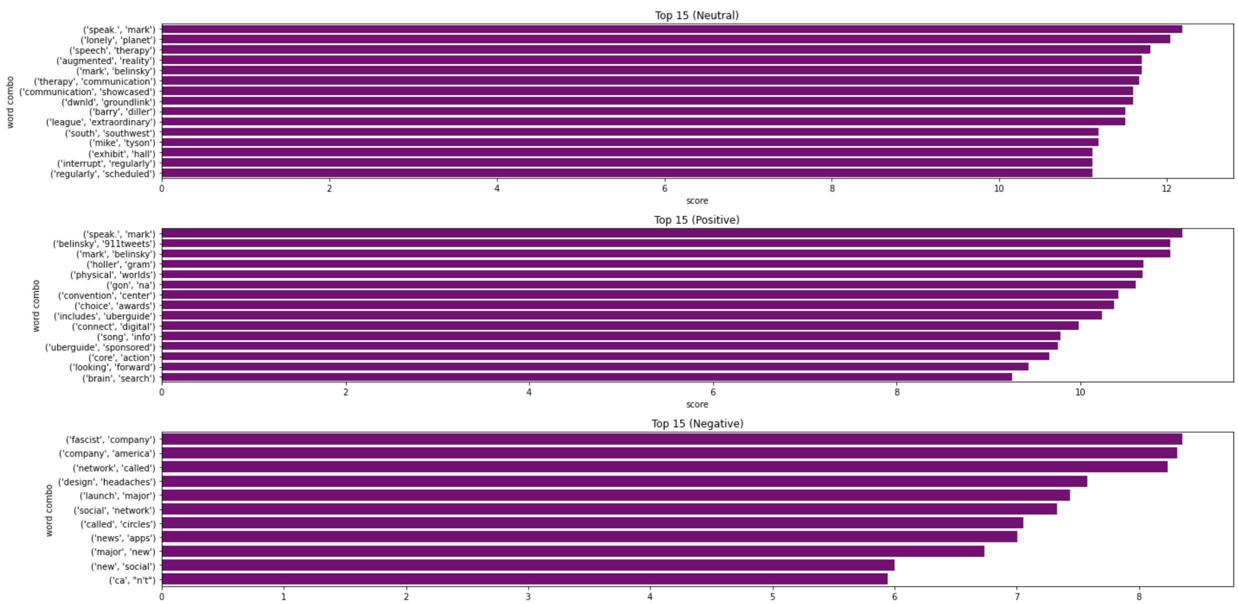


Similar to the frequency distributions, we see a number of the same results showing up commonly among the different class labels. We will move on to show the PMI for each class. Some bigrams stand out among the negative class including: ('apps', 'fades'), ('fades', 'fast').

In [42]:
```
print('Neutral PMI:')
neutral_pmi = display_pmi(concat_neutral)
print('Positive PMI:')
positive_pmi = display_pmi(concat_positive)
print('Negative PMI:')
negative_pmi = display_pmi(concat_negative)
```

```
Neutral PMI:

[(('speak.', 'mark'), 12.180219982170193),
 (('lonely', 'planet'), 12.042716458420257),
 (('speech', 'therapy'), 11.801708358916462),
 (('augmented', 'reality'), 11.69479315499995),
 (('mark', 'belinsky'), 11.69479315499995),
 (('therapy', 'communication'), 11.664204835166528),
 (('communication', 'showcased'), 11.595257481449035),
 (('dwnld', 'groundlink'), 11.595257481449035),
 (('barry', 'diller'), 11.502148077057555),
 (('league', 'extraordinary'), 11.502148077057551),
 (('south', 'southwest'), 11.180219982170192),
 (('mike', 'tyson'), 11.18021998217019),
 (('exhibit', 'hall'), 11.109830654278795),
 (('interrupt', 'regularly'), 11.109830654278793),
 (('regularly', 'scheduled'), 11.109830654278793)]

Positive PMI:

[(('speak.', 'mark'), 11.107435054747327),
 (('belinsky', '911tweets'), 10.98190417266347),
 (('mark', 'belinsky'), 10.98190417266347),
 (('holler', 'gram'), 10.692397555468485),
 (('physical', 'worlds'), 10.678591755943454),
 (('gon', 'na'), 10.604934714218146),
 (('convention', 'center'), 10.412289636275752),
 (('choice', 'awards'), 10.370469460581122),
 (('includes', 'uberguide'), 10.232965936831187),
 (('connect', 'digital'), 9.978152037802364),
 (('song', 'info'), 9.785506959859966),
 (('uberguide', 'sponsored'), 9.759511751327022),
 (('core', 'action'), 9.660198733080337),
 (('looking', 'forward'), 9.435009712775834),
 (('brain', 'search'), 9.259438148192379)]

Negative PMI:

[(('fascist', 'company'), 8.35395694908),
 (('company', 'america'), 8.313314964582656),
 (('network', 'called'), 8.23580559736174),
 (('design', 'headaches'), 7.57634937041645),
 (('launch', 'major'), 7.4388458466665135),
 (('social', 'network'), 7.329972308536261),
 (('called', 'circles'), 7.055233351719918),
 (('news', 'apps'), 7.007328413564313),
 (('major', 'new'), 6.735047116435506),
 (('new', 'social'), 6.003243227385081),
 (('ca', "n't"), 5.940760796625325)]
```

In [43]:
```
# pull out top n pmi scores
top_n = 15
top_neutral_pmi = pd.DataFrame(neutral_pmi[:top_n], columns=['word combo',
top_positive_pmi = pd.DataFrame(positive_pmi[:top_n], columns=['word combo'
top_negative_pmi = pd.DataFrame(negative_pmi[:top_n], columns=['word combo'

# visualize
fig, axes = plt.subplots(3, 1, figsize=(20, 10))
pmis = [top_neutral_pmi, top_positive_pmi, top_negative_pmi]
labels = ['Top 15 (Neutral)', 'Top 15 (Positive)', 'Top 15 (Negative)', 'To
for idx, ax in enumerate(axes.flat):
    sns.barplot(data=pmis[idx],
                y='word combo',
                x='score',
                ax=ax,
                color='purple')
    ax.set_title(labels[idx])
plt.tight_layout()
```



Looking at PMI scores, we can see some further trends standing out. Some word combinations within the positive dataset that stand out include: (choice, awards), (uberguide, sponsored), (looking, forward). Some word combinations that stood out within the negative set included: (fascist, company) and (design, headaches).

Now that we have a good sense of our data and the distribution, we can move on to the data preparation phase

3. Data Preparation

Leverage information learned during data understanding phase to preprocess dataset and prepare data for modeling.

In [44]:
```
# set seed for reproducibility
SEED = 1
```

In [45]:
```
raw_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9093 entries, 0 to 9092
Data columns (total 3 columns):
 #   Column                                              Non-Null Count
Dtype
---  ------                                              --------------
-----
 0   tweet_text                                          9092 non-null
object
 1   emotion_in_tweet_is_directed_at                     3291 non-null
object
 2   is_there_an_emotion_directed_at_a_brand_or_product  9093 non-null
object
dtypes: object(3)
memory usage: 213.2+ KB
```

In [46]:
```
# pull in copy of dataset
clean_df = raw_df.copy()

# relabel columns
clean_df.columns = ['text', 'product_brand', 'sentiment']

# drop product_brand column, handle missing values and duplicates
clean_df = clean_df.drop('product_brand', axis=1)
clean_df = clean_df.dropna()
clean_df = clean_df.drop_duplicates()

# remove ambiguous tweets
clean_df = clean_df.loc[clean_df['sentiment'] != "I can't tell"]
```

In [47]:
```
# separate dataset into text and class_labels
text = clean_df['text']
class_labels = clean_df['sentiment']
```

In [48]:
```
# split tweets and labels into train and test sets for validation purposes
X_train, X_test, y_train, y_test = train_test_split(text, class_labels, str
```

In [49]:
```
# update stopwords list per our data understanding findings
updated_stopwords = stopwords_list + additional_stopwords
```

In [50]:
```
def preprocess_tweet(tweet, stopwords_list):
    """
    Function to preprocess a tweet.
    Takes: tweet, stopwords list
    Returns: processed tweet with stopwords removed and converted to lowerc
    """
    processed = re.sub("\'", '', tweet) # handle apostrophes
    processed = re.sub('\s+', ' ', processed) # handle excess white space
    tokens = nltk.word_tokenize(processed)
    stopwords_removed = [token.lower() for token in tokens if token.lower()
    return ' '.join(stopwords_removed)
```

In [51]:
```
# preprocess train and test sets
X_train_preprocessed = X_train.apply(lambda x: preprocess_tweet(x, updated_
X_test_preprocessed = X_test.apply(lambda x: preprocess_tweet(x, updated_st
```

Now that we have split our data into train and test sets, as well as, preprocessed both train and test sets, we are ready to vectorize our data. We have chosen to use TF-IDF vectorization for its benefits in classification and finding words that are unique per class label. Try a number of vectorizers to see if there is one that performs better over the other (count vectorized vs. TF-IDF vectorized).

In [52]:
```
# create vectorizers with unigram and bigrams
count_vectorizer = CountVectorizer(ngram_range=(1,2), analyzer='word') # us
tfidf_vectorizer = TfidfVectorizer(ngram_range=(1,2), analyzer='word') # us

# fit to preprocessed data
X_train_count = count_vectorizer.fit_transform(X_train_preprocessed)
X_test_count = count_vectorizer.transform(X_test_preprocessed)
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train_preprocessed)
X_test_tfidf = tfidf_vectorizer.transform(X_test_preprocessed)
```

Word Embeddings

In [53]:
```
# tokenize datasets
tokenized_X_train = X_train_preprocessed.map(nltk.word_tokenize).values
tokenized_X_test = X_test_preprocessed.map(nltk.word_tokenize).values

# get total training vocabulary size
total_train_vocab = set(word for tweet in tokenized_X_train for word in twe
train_vocab_size = len(total_train_vocab)
print(f'There are {train_vocab_size} unique tokens in the processed trainin
```

```
There are 8951 unique tokens in the processed training set.
```

We will take advantage of Global Vectors for Word Representation (GloVe). For more information please visit [https://nlp.stanford.edu/projects/glove/ (https://nlp.stanford.edu/projects/glove/)](https://nlp.stanford.edu/projects/glove/)

In [54]:
```
def glove_vectors(vocab):
    """
    Returns appropriate vectors from GloVe file.
    Input: vocabulary set to use.
    """
    glove = {}
    with open('glove.6B.50d.txt', 'rb') as f:
        for line in f:
            parts = line.split()
            word = parts[0].decode('utf-8')
            if word in vocab:
                vector = np.array(parts[1:], dtype=np.float32)
                glove[word] = vector
    return glove
```

In [55]:
```
glove = glove_vectors(total_train_vocab)
```

In [56]:
```
class W2vVectorizer(object):

    def __init__(self, w2v):
        # Takes in a dictionary of words and vectors as input
        self.w2v = w2v
        if len(w2v) == 0:
            self.dimensions = 0
        else:
            self.dimensions = len(w2v[next(iter(glove))])

    # Note: Even though it doesn't do anything, it's required that this obj
    # it can't be used in a scikit-learn pipeline
    def fit(self, X, y):
        return self

    def transform(self, X):
        return np.array([
            np.mean([self.w2v[w] for w in words if w in self.w2v]
                    or [np.zeros(self.dimensions)], axis=0) for words in X])
```

In [57]:
```
# instantiate vectorizer objects with glove
w2v_vectorizer = W2vVectorizer(glove)

# transform training and testing data
X_train_w2v = w2v_vectorizer.transform(tokenized_X_train)
X_test_w2v = w2v_vectorizer.transform(tokenized_X_test)
```

Now that we have vectorized our datasets, we are ready to move on to the modeling stage.

4. Modeling

This is a classification task, tasked with classifying the sentiment of tweets based on the text within the tweet. Three primary models will be relied on for classification:

1. Random Forests
2. Linear SVM
3. Neural Networks

Overfitting will be addressed thru hyperparameter tuning and dropout layers in the case of neural networks.

This is a multi-class classification problem, with three available class labels (Neutral, Positive, or Negative). As a result, the performance metric we will focus on throughout this process will be accuracy. We are not too concerned about the ramifications of false positives or false negatives. For this reason accuracy will be our selected perforamnce metric.

Random Forest

In [58]:
```
# instantiate random forest classifiers, with balanced class_weight
rf_count = RandomForestClassifier(random_state=SEED, n_jobs=-1, class_weigh
rf_tfidf = RandomForestClassifier(random_state=SEED, n_jobs=-1, class_weigh
rf_w2v = RandomForestClassifier(random_state=SEED, n_jobs=-1, class_weight=

# fit to training sets
rf_count.fit(X_train_count, y_train)
rf_tfidf.fit(X_train_tfidf, y_train)
rf_w2v.fit(X_train_w2v, y_train)
```

Out[58]:
```
RandomForestClassifier(class_weight='balanced', n_jobs=-1, random_state=
1)
```

In [59]:
```
# Count Vectorized
count_train_score = rf_count.score(X_train_count, y_train)
count_test_score = rf_count.score(X_test_count, y_test)
print(f'Count Vectorized Train Score: {count_train_score}')
print(f'Count Vectorized Test Score: {count_test_score}')
print('--------')

# TF-IDF Vectorized
tfidf_train_score = rf_tfidf.score(X_train_tfidf, y_train)
tfidf_test_score = rf_tfidf.score(X_test_tfidf, y_test)
print(f'TF-IDF Vectorized Train Score: {tfidf_train_score}')
print(f'TF-IDF Vectorized Test Score: {tfidf_test_score}')
print('--------')

# W2V Vectorized
w2v_train_score = rf_w2v.score(X_train_w2v, y_train)
w2v_test_score = rf_w2v.score(X_test_w2v, y_test)
print(f'Word2Vec Vectorized Train Score: {w2v_train_score}')
print(f'Word2Vec Vectorized Test Score: {w2v_test_score}')
```

```
Count Vectorized Train Score: 0.9603590127150337
Count Vectorized Test Score: 0.6778824585015703
--------
TF-IDF Vectorized Train Score: 0.9603590127150337
TF-IDF Vectorized Test Score: 0.6774338268281741
--------
Word2Vec Vectorized Train Score: 0.9545250560957367
Word2Vec Vectorized Test Score: 0.6514131897711979
```

Reviewing baseline random model scores for our three vectorized datasets (count, tf-idf, and word2vec using glove), we can see that results are fairly consistent across our vectorization methods. Further, looking at our high training set accuracy score vs. test scores, shows we are likely overfitting slightly to the training data.

Address overfitting thru hyperparameter tuning.

Random Forest - Count Vectorized

Tune hyperparams with grid search

In [60]:
```
# set params
grid_search_params = {
    'min_samples_split': [4, 5],
    'min_samples_leaf': [3, 4],
    'max_depth': [25, 50, 75],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False],
    'criterion': ['entropy', 'gini']
}

# instantiate classifier
rf_classifier = RandomForestClassifier(n_jobs=-1, random_state=SEED, class_

# instantiate grid search
rf_gs_count = GridSearchCV(estimator=rf_classifier,
                           param_grid=grid_search_params,
                           cv=3,
                           scoring='accuracy',
                           return_train_score=True,
                           verbose=1)
```

In [61]:
```
# fit to count vectorized
rf_gs_count.fit(X_train_count, y_train)
```

```
Fitting 3 folds for each of 96 candidates, totalling 288 fits
```

Out[61]:
```
GridSearchCV(cv=3,
             estimator=RandomForestClassifier(class_weight='balanced',
                                              n_jobs=-1, random_state=1),
             param_grid={'bootstrap': [True, False],
                         'criterion': ['entropy', 'gini'],
                         'max_depth': [25, 50, 75],
                         'max_features': ['auto', 'sqrt'],
                         'min_samples_leaf': [3, 4],
                         'min_samples_split': [4, 5]},
             return_train_score=True, scoring='accuracy', verbose=1)
```

In [62]:
```
# print count-vectorized results
mean_train_score_count = np.mean(rf_gs_count.cv_results_['mean_train_score'
mean_test_score_count = np.mean(rf_gs_count.cv_results_['mean_test_score'])
print(f'Random Search Train Accuracy (Count Vect.): {mean_train_score_count
print(f'Random Search Test Accuracy (Count Vect.): {mean_test_score_count}'

# display best params
rf_gs_count.best_params_
```

```
Random Search Train Accuracy (Count Vect.): 0.6580837817803394
Random Search Test Accuracy (Count Vect.): 0.5654601087866924
```

Out[62]:
```
{'bootstrap': True,
 'criterion': 'gini',
 'max_depth': 75,
 'max_features': 'auto',
 'min_samples_leaf': 3,
 'min_samples_split': 4}
```

We have successfully addressed overfitting, as evidenced by closeness of training and testing accuracy. Fit model with these params and reprint training and testing scores.

In [63]:
```
# run best count-vect model with these params
best_rf_count = RandomForestClassifier(class_weight='balanced', n_jobs=-1,
                                       bootstrap=True,
                                       criterion='gini',
                                       max_depth=75,
                                       max_features='auto',
                                       min_samples_leaf=3,
                                       min_samples_split=4)

# fit to count vect data
best_rf_count.fit(X_train_count, y_train)

# print testing score
print(f'Best Tuned Random Forest (Count Vectorized) Test Accuracy: {best_rf
print(f'Best Tuned Random Forest (Count Vectorized) Train Accuracy: {best_r
```

```
Best Tuned Random Forest (Count Vectorized) Test Accuracy: 0.608344549125
1682
Best Tuned Random Forest (Count Vectorized) Train Accuracy: 0.68706058339
56619
```

Looking at the best identified tuned random forest model on our count vectorized data, we see overfitting has largely been addressed, but model performance is not looking great with testing accuracy scores coming in close to ~61%.

Random Forest - TF-IDF Vectorized

Use grid search to tune params. Still need to address overfitting identified in baseline models.

In [64]:
```
# similar to count vectorized, run refined grid search and try to further ad
grid_search_params = {
    'min_samples_split': [4, 5],
    'min_samples_leaf': [3, 4],
    'max_depth': [25, 50, 75],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False],
    'criterion': ['entropy', 'gini']
}

# instantiate classifier
rf_classifier = RandomForestClassifier(n_jobs=-1, random_state=SEED, class_w

# instantiate grid search
rf_gs_tfidf = GridSearchCV(estimator=rf_classifier,
                           param_grid=grid_search_params,
                           cv=3,
                           scoring='accuracy',
                           return_train_score=True,
                           verbose=1)
```

In [65]:
```
# fit to training data
rf_gs_tfidf.fit(X_train_tfidf, y_train)
```

```
Fitting 3 folds for each of 96 candidates, totalling 288 fits
```

Out[65]:
```
GridSearchCV(cv=3,
             estimator=RandomForestClassifier(class_weight='balanced',
                                              n_jobs=-1, random_state=1),
             param_grid={'bootstrap': [True, False],
                         'criterion': ['entropy', 'gini'],
                         'max_depth': [25, 50, 75],
                         'max_features': ['auto', 'sqrt'],
                         'min_samples_leaf': [3, 4],
                         'min_samples_split': [4, 5]},
             return_train_score=True, scoring='accuracy', verbose=1)
```

In [66]:
```
# print search results
mean_train_score_tfidf = np.mean(rf_gs_tfidf.cv_results_['mean_train_score'
mean_test_score_tfidf = np.mean(rf_gs_tfidf.cv_results_['mean_test_score'])
print(f'Grid Search Train Accuracy (TF-IDF): {mean_train_score_tfidf}')
print(f'Grid Search Test Accuracy (TF-IDF): {mean_test_score_tfidf}')

# display best params
rf_gs_tfidf.best_params_
```

```
Grid Search Train Accuracy (TF-IDF): 0.6774899585455202
Grid Search Test Accuracy (TF-IDF): 0.561869738164845
```

Out[66]:
```
{'bootstrap': True,
 'criterion': 'gini',
 'max_depth': 75,
 'max_features': 'auto',
 'min_samples_leaf': 3,
 'min_samples_split': 4}
```

Similar to count vectorized data, we see overfitting has been addressed through tuning of hyperparams. Move forward with fitting a model with these params and refitting to training data.

In [67]:
```
# run best tfidf model with identified params
best_rf_tfidf = RandomForestClassifier(random_state=SEED, n_jobs=-1, class_
                                       min_samples_split=4,
                                       min_samples_leaf=3,
                                       max_depth=75,
                                       max_features='auto',
                                       bootstrap=True,
                                       criterion='gini')

# fit to tfidf vectorized
best_rf_tfidf.fit(X_train_tfidf, y_train)

# print testing score
print(f'Best Tuned Random Forest (TF-IDF Vectorized) Test Accuracy: {best_r
print(f'Best Tuned Random Forest (TF-IDF Vectorized) Train Accuracy: {best_
```

```
Best Tuned Random Forest (TF-IDF Vectorized) Test Accuracy: 0.59712875729
02647
Best Tuned Random Forest (TF-IDF Vectorized) Train Accuracy: 0.7108451757
666417
```

Looking at the best identified tuned random forest results on the TF-IDF vectorized dataset, we can see results are similar to those seen with count vectorized data. Although we have addressed overfitting to training data that was present in our baselines, accuracy scores are still hovering around 60%. Move forward with next vectorization strategy

Random Forest - Word2Vec Vectorized

In [68]:
```
# further refine with grid search and further address overfitting
rf_params_w2v = {
    'min_samples_split': [5, 6],
    'min_samples_leaf': [3, 4],
    'max_depth': [3, 4, 5],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False],
    'criterion': ['entropy', 'gini']
}

# instantiate classifier
rf_classifier = RandomForestClassifier(n_jobs=-1, random_state=SEED, class_

# instantiate grid search
rf_gs_w2v = GridSearchCV(estimator=rf_classifier,
                         param_grid=rf_params_w2v,
                         cv=3,
                         scoring='accuracy',
                         return_train_score=True,
                         verbose=1)
```

In [69]:
```
rf_gs_w2v.fit(X_train_w2v, y_train)
```

```
Fitting 3 folds for each of 96 candidates, totalling 288 fits
```

Out[69]:
```
GridSearchCV(cv=3,
             estimator=RandomForestClassifier(class_weight='balanced',
                                              n_jobs=-1, random_state=1),
             param_grid={'bootstrap': [True, False],
                         'criterion': ['entropy', 'gini'],
                         'max_depth': [3, 4, 5],
                         'max_features': ['auto', 'sqrt'],
                         'min_samples_leaf': [3, 4],
                         'min_samples_split': [5, 6]},
             return_train_score=True, scoring='accuracy', verbose=1)
```

In [70]:
```
# print word2vec vectorized grid search results
mean_train_score_w2v = np.mean(rf_gs_w2v.cv_results_['mean_train_score'])
mean_test_score_w2v = np.mean(rf_gs_w2v.cv_results_['mean_test_score'])
print(f'Grid Search Train Accuracy (word2vec): {mean_train_score_w2v}')
print(f'Grid Search Test Accuracy (word2vec): {mean_test_score_w2v}')

# display best params
rf_gs_w2v.best_params_
```

```
Grid Search Train Accuracy (word2vec): 0.5597883850373593
Grid Search Test Accuracy (word2vec): 0.48973472862437434
```

Out[70]:
```
{'bootstrap': True,
 'criterion': 'gini',
 'max_depth': 5,
 'max_features': 'auto',
 'min_samples_leaf': 3,
 'min_samples_split': 5}
```

While overfitting has been addressed, overall model performance is notably worse than count and TF-IDF vectorized data. Save a version of this as the best random forest we saw with w2v. Given low performance, this model will most likely not be selected.

In [71]:
```
# fit model using best found params for w2v data
best_rf_w2v = RandomForestClassifier(random_state=SEED, class_weight='balan
                                     n_estimators=75, bootstrap=True, crite
                                     max_depth=5, max_features='auto', min_
                                     min_samples_split=5)

best_rf_w2v.fit(X_train_w2v, y_train)

# print testing score
print(f'Best Tuned Random Forest (Word2Vec) Test Accuracy: {best_rf_w2v.sco
print(f'Best Tuned Random Forest (Word2Vec) Train Accuracy: {best_rf_w2v.sc
```

```
Best Tuned Random Forest (Word2Vec) Test Accuracy: 0.5074024226110363
Best Tuned Random Forest (Word2Vec) Train Accuracy: 0.5968586387434555
```

Using our word2vec vectorized data, we can see that overfitting has largely been addressed, but results are weaker than both count and TF-IDF vectorized data, with testing accuracy coming in around ~50%.

Linear SVM

Move on to modeling with LinearSVC classifier.

In [72]:
```
# create linear SVC
svc_count = LinearSVC(random_state=SEED, class_weight='balanced', max_iter=
svc_tfidf = LinearSVC(random_state=SEED, class_weight='balanced', max_iter=
svc_w2v = LinearSVC(random_state=SEED, class_weight='balanced', max_iter=50

# fit to training sets
svc_count.fit(X_train_count, y_train)
svc_tfidf.fit(X_train_tfidf, y_train)
svc_w2v.fit(X_train_w2v, y_train)
```

Out[72]:
```
LinearSVC(class_weight='balanced', max_iter=5000, random_state=1)
```

In [73]:
```
# Count Vectorized
count_train_score = svc_count.score(X_train_count, y_train)
count_test_score = svc_count.score(X_test_count, y_test)
print(f'Count Vectorized Train Score: {count_train_score}')
print(f'Count Vectorized Test Score: {count_test_score}')
print('-----')

# TF-IDF Vectorized
tfidf_train_score = svc_tfidf.score(X_train_tfidf, y_train)
tfidf_test_score = svc_tfidf.score(X_test_tfidf, y_test)
print(f'TF-IDF Vectorized Train Score: {tfidf_train_score}')
print(f'TF-IDF Vectorized Test Score: {tfidf_test_score}')
print('-----')

# Word2Vec Vectorized
w2v_train_score = svc_w2v.score(X_train_w2v, y_train)
w2v_test_score = svc_w2v.score(X_test_w2v, y_test)
print(f'Word2Vect Vectorized Train Score: {w2v_train_score}')
print(f'Word2Vect Vectorized Test Score: {w2v_test_score}')
```

```
Count Vectorized Train Score: 0.9605086013462977
Count Vectorized Test Score: 0.686855091969493
-----
TF-IDF Vectorized Train Score: 0.9543754674644727
TF-IDF Vectorized Test Score: 0.6913414087034545
-----
Word2Vect Vectorized Train Score: 0.5992520568436799
Word2Vect Vectorized Test Score: 0.5895020188425303
```

Looking at baseline LinearSVC results we can see that, similar to initial random forest models, we are overfitting to training data, except with word2vec vectorized data. Move forward with hyperparameter tuning to try and improve results / address overfitting

LinearSVC - Count Vectorized

In [74]:
```
# set params for grid search
svc_params = {
    'C': [0.00001, 0.0001, .001],
    'loss': ['hinge', 'squared_hinge']
}
```

In [75]:
```
# grid search
svc_classifier = LinearSVC(random_state=SEED, class_weight='balanced', max_
svc_gs_count = GridSearchCV(svc_classifier,
                            svc_params,
                            return_train_score=True,
                            scoring='accuracy',
                            verbose=1)
```

In [76]:
```
# fit to training data
svc_gs_count.fit(X_train_count, y_train)
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
```

Out[76]:
```
GridSearchCV(estimator=LinearSVC(class_weight='balanced', max_iter=10000,
                                 random_state=1),
             param_grid={'C': [1e-05, 0.0001, 0.001],
                         'loss': ['hinge', 'squared_hinge']},
             return_train_score=True, scoring='accuracy', verbose=1)
```

In [77]:
```
# print count-vectorized grid-search results
mean_train_score_count = np.mean(svc_gs_count.cv_results_['mean_train_score
mean_test_score_count = np.mean(svc_gs_count.cv_results_['mean_test_score']
print(f'Grid Search Train Accuracy (Count Vect.): {mean_train_score_count}'
print(f'Grid Search Test Accuracy (Count Vect.): {mean_test_score_count}')

# display best params
svc_gs_count.best_params_
```

```
Grid Search Train Accuracy (Count Vect.): 0.6216778858140115
Grid Search Test Accuracy (Count Vect.): 0.6111692844677138
```

Out[77]:
```
{'C': 0.001, 'loss': 'squared_hinge'}
```

Through tuning, we have addressed overfitting, move forward with fitting a model with the identified params and fit to full training set.

In [78]:
```
# best count svc
best_svc_count = LinearSVC(random_state=SEED, class_weight='balanced', max_
                           C=0.001, loss='squared_hinge')

best_svc_count.fit(X_train_count, y_train)

# print testing score
print(f'Best Tuned Linear SVC (Count) Test Accuracy: {best_svc_count.score(
print(f'Best Tuned Linear SVC (Count) Train Accuracy: {best_svc_count.score
```

```
Best Tuned Linear SVC (Count) Test Accuracy: 0.65814266487214
Best Tuned Linear SVC (Count) Train Accuracy: 0.7123410620792819
```

Best identified LinearSVC test accuracy coming in around 66%, and training accuracy around 71%. Fairly strong results with quick runtimes. Strong contender for best model so far.

Linear SVC (TF-IDF)

In [79]:
```
# failing to converge, try with tfidf vectorized data
# set params
svc_params = {
    'C': [0.0001, .001, 0.01],
    'loss': ['hinge', 'squared_hinge']
}

# grid search
svc_classifier = LinearSVC(random_state=SEED, class_weight='balanced', max_
svc_gs_tfidf = GridSearchCV(svc_classifier,
                            svc_params,
                            return_train_score=True,
                            scoring='accuracy')

svc_gs_tfidf.fit(X_train_tfidf, y_train)
```

Out[79]:
```
GridSearchCV(estimator=LinearSVC(class_weight='balanced', max_iter=10000,
                                 random_state=1),
             param_grid={'C': [0.0001, 0.001, 0.01],
                         'loss': ['hinge', 'squared_hinge']},
             return_train_score=True, scoring='accuracy')
```

In [80]:
```
# print tfidf-vectorized grid search results
mean_train_score_tfidf = np.mean(svc_gs_tfidf.cv_results_['mean_train_score
mean_test_score_tfidf = np.mean(svc_gs_tfidf.cv_results_['mean_test_score']
print(f'Grid Search Train Accuracy (TF-IDF): {mean_train_score_tfidf}')
print(f'Grid Search Test Accuracy (TF-IDF): {mean_test_score_tfidf}')

# display best params
svc_gs_tfidf.best_params_
```

```
Grid Search Train Accuracy (TF-IDF): 0.650330341560708
Grid Search Test Accuracy (TF-IDF): 0.6151583146347545
```

Out[80]:
```
{'C': 0.01, 'loss': 'hinge'}
```

Again given similarities in testing and training data, we have likely addressed overfitting. However, results are not looking great, especially when compared to strong results seen from count vectorized data.

In [81]:
```
# fit best svc for tfidf
best_svc_tfidf = LinearSVC(random_state=SEED, class_weight='balanced', max_
                           C=0.01, loss='squared_hinge')

best_svc_tfidf.fit(X_train_tfidf, y_train)

# print testing score
print(f'Best Tuned Linear SVC (TF-IDF) Test Accuracy: {best_svc_tfidf.score
print(f'Best Tuned Linear SVC (TF-IDF) Train Accuracy: {best_svc_tfidf.scor
```

```
Best Tuned Linear SVC (TF-IDF) Test Accuracy: 0.6294302377747869
Best Tuned Linear SVC (TF-IDF) Train Accuracy: 0.630964846671653
```

Best identified Linear SVC with TF-IDF vectorization found testing score around 63% and training score around 63%. Performance is fairly strong, with overfitting being addressed. Still underperforming count vectorized. Move on to Word2Vec vectorization.

LinearSVC (Word2Vec)

In [82]:
```
svc_params = {
    'C': [0.01, 0.1, 1],
    'loss': ['hinge', 'squared_hinge']
}

# grid search
svc_classifier = LinearSVC(random_state=SEED, class_weight='balanced', max_
svc_gs_w2v = GridSearchCV(svc_classifier,
                          svc_params,
                          return_train_score=True,
                          scoring='accuracy',
                          verbose=1)

# fit to tfidf data
svc_gs_w2v.fit(X_train_w2v, y_train)
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits

/Users/addingtongraham/opt/anaconda3/envs/keras-env/lib/python3.6/site-pa
ckages/sklearn/svm/_base.py:986: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/addingtongraham/opt/anaconda3/envs/keras-env/lib/python3.6/site-pa
ckages/sklearn/svm/_base.py:986: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/addingtongraham/opt/anaconda3/envs/keras-env/lib/python3.6/site-pa
ckages/sklearn/svm/_base.py:986: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/addingtongraham/opt/anaconda3/envs/keras-env/lib/python3.6/site-pa
ckages/sklearn/svm/_base.py:986: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/addingtongraham/opt/anaconda3/envs/keras-env/lib/python3.6/site-pa
ckages/sklearn/svm/_base.py:986: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/addingtongraham/opt/anaconda3/envs/keras-env/lib/python3.6/site-pa
ckages/sklearn/svm/_base.py:986: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/addingtongraham/opt/anaconda3/envs/keras-env/lib/python3.6/site-pa
ckages/sklearn/svm/_base.py:986: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
```

Out[82]:
```
GridSearchCV(estimator=LinearSVC(class_weight='balanced', max_iter=10000,
                                 random_state=1),
             param_grid={'C': [0.01, 0.1, 1],
                         'loss': ['hinge', 'squared_hinge']},
             return_train_score=True, scoring='accuracy', verbose=1)
```

In [83]:
```
# print tfidf-vectorized grid search results
mean_train_score_w2v = np.mean(svc_gs_w2v.cv_results_['mean_train_score'])
mean_test_score_w2v = np.mean(svc_gs_w2v.cv_results_['mean_test_score'])
print(f'Grid Search Train Accuracy (Word2Vec): {mean_train_score_w2v}')
print(f'Grid Search Test Accuracy (Word2Vec): {mean_test_score_w2v}')

# display best params
svc_gs_w2v.best_params_
```

```
Grid Search Train Accuracy (Word2Vec): 0.586786337571678
Grid Search Test Accuracy (Word2Vec): 0.5796559461480928
```

Out[83]:
```
{'C': 1, 'loss': 'squared_hinge'}
```

Minimal overfitting, params have not changed since baseline, although performance is fairly low.

In [84]:
```
# run best svc with w2v
best_svc_w2v = LinearSVC(random_state=SEED,
                         class_weight='balanced',
                         max_iter=10000,
                         C=1,
                         loss='squared_hinge')

best_svc_w2v.fit(X_train_w2v, y_train)

# print testing score
print(f'Best Tuned Linear SVC (Word2Vec) Test Accuracy: {best_svc_w2v.score
print(f'Best Tuned Linear SVC (Word2Vec) Train Accuracy: {best_svc_w2v.scor
```

```
Best Tuned Linear SVC (Word2Vec) Test Accuracy: 0.5895020188425303
Best Tuned Linear SVC (Word2Vec) Train Accuracy: 0.5992520568436799
```

Best identified tuned LinearSVC using word2vec data is not changed from baseline and is still showing ~59% accuracy score. This is performing worse than the other two vectorization methods. Move forward with Neural Network Modeling.

Neural Networks

In [85]:
```
# set random states for neural network reproducibility
import tensorflow
tensorflow.random.set_seed(SEED)
```

In [86]:
```
# convert labels to one-hot encoded format
y_train_encoded = pd.get_dummies(y_train).values
y_test_encoded = pd.get_dummies(y_test).values
```

In [87]:
```
# set up last layer of neural network for multi-class classification with 3
last_layer_activation = 'softmax'
last_layer_units = 3
```

N-Gram Model

In [88]:
```
def build_ngram_model(input_shape, last_units, last_activation):
    """
    builds and compiles model based on input params.
    returns model.
    """

    # build model
    model = Sequential()
    model.add(Dropout(rate=0.5, input_shape=input_shape))
    model.add(Dense(units=50, activation='relu'))
    model.add(Dropout(rate=0.5))
    model.add(Dense(units=50, activation='relu'))
    model.add(Dropout(rate=0.5))
    model.add(Dense(units=last_units, activation=last_activation))

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

    # display model summary
    display(model.summary())

    return model
```

N-Gram Neural Network (Count)

In [89]:
```
# build and compile
count_model = build_ngram_model(input_shape=X_train_count.shape[1:],
                                last_units=last_layer_units,
                                last_activation=last_layer_activation)
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout_1 (Dropout)          (None, 38654)             0
_________________________________________________________________
dense_1 (Dense)              (None, 50)                1932750
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_2 (Dense)              (None, 50)                2550
_________________________________________________________________
dropout_3 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 153
=================================================================
Total params: 1,935,453
Trainable params: 1,935,453
Non-trainable params: 0
_________________________________________________________________

None
```

In [90]:
```python
# fit to count vectorized
count_model.fit(X_train_count, y_train_encoded,
                epochs=10,
                batch_size=100,
                validation_split=0.2)
```

```
Train on 5348 samples, validate on 1337 samples
Epoch 1/10
5348/5348 [==============================] - 7s 1ms/step - loss: 0.9720 -
accuracy: 0.5884 - val_loss: 0.8359 - val_accuracy: 0.6028
Epoch 2/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.7983 -
accuracy: 0.6212 - val_loss: 0.7784 - val_accuracy: 0.6417
Epoch 3/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.6958 -
accuracy: 0.6948 - val_loss: 0.7527 - val_accuracy: 0.6597
Epoch 4/10
5348/5348 [==============================] - 7s 1ms/step - loss: 0.5741 -
accuracy: 0.7693 - val_loss: 0.7678 - val_accuracy: 0.6649
Epoch 5/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.4683 -
accuracy: 0.8164 - val_loss: 0.8105 - val_accuracy: 0.6545
Epoch 6/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.3986 -
accuracy: 0.8470 - val_loss: 0.8427 - val_accuracy: 0.6582
Epoch 7/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.3362 -
accuracy: 0.8794 - val_loss: 0.8950 - val_accuracy: 0.6515
Epoch 8/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.3021 -
accuracy: 0.8902 - val_loss: 0.9289 - val_accuracy: 0.6462
Epoch 9/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.2854 -
accuracy: 0.8968 - val_loss: 0.9664 - val_accuracy: 0.6530
Epoch 10/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.2523 -
accuracy: 0.9095 - val_loss: 0.9850 - val_accuracy: 0.6440
```

Out[90]:
```
<keras.callbacks.callbacks.History at 0x7f97b92a2a58>
```

Looking at the neural network we can see that after epoch 4, the model starts overfitting to the training data. Stopping after epoch 3 will help control overfitting to training data.

In [91]:
```
# build and compile
count_model = build_ngram_model(input_shape=X_train_count.shape[1:],
                                last_units=last_layer_units,
                                last_activation=last_layer_activation)

# train with 3 epochs
count_model.fit(X_train_count, y_train_encoded,
                epochs=3,
                batch_size=100,
                validation_split=0.2)
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout_4 (Dropout)          (None, 38654)             0
_________________________________________________________________
dense_4 (Dense)              (None, 50)                1932750
_________________________________________________________________
dropout_5 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_5 (Dense)              (None, 50)                2550
_________________________________________________________________
dropout_6 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_6 (Dense)              (None, 3)                 153
=================================================================
Total params: 1,935,453
Trainable params: 1,935,453
Non-trainable params: 0
_________________________________________________________________
None
Train on 5348 samples, validate on 1337 samples
Epoch 1/3
5348/5348 [==============================] - 7s 1ms/step - loss: 0.9761 - accuracy: 0.5623 - val_loss: 0.8331 - val_accuracy: 0.6028
Epoch 2/3
5348/5348 [==============================] - 6s 1ms/step - loss: 0.7964 - accuracy: 0.6309 - val_loss: 0.7752 - val_accuracy: 0.6395
Epoch 3/3
5348/5348 [==============================] - 6s 1ms/step - loss: 0.6896 - accuracy: 0.7010 - val_loss: 0.7475 - val_accuracy: 0.6612
```

Out[91]:
```
<keras.callbacks.callbacks.History at 0x7f97aef8b588>
```

In [92]:
```
_, nn_count_train_score = count_model.evaluate(X_train_count, y_train_encod
_, nn_count_test_score = count_model.evaluate(X_test_count, y_test_encoded,

print(f'Train Accuracy (Neural Network / Count Vectorized): {nn_count_train
print(f'Test Accuracy (Neural Network / Count Vectorized): {nn_count_test_s
```

```
Train Accuracy (Neural Network / Count Vectorized): 0.7880328893661499
Test Accuracy (Neural Network / Count Vectorized): 0.6680125594139099
```

With ~67% testing accuracy and 77% training accuracy, this model is our best performer yet.

Given fairly similar testing and training scores, our model is not likely overfitting to the training set much.

N-Gram Neural Network (TF-IDF)

In [93]:
```
# build and compile
tfidf_model = build_ngram_model(input_shape=X_train_tfidf.shape[1:],
                                last_units=last_layer_units,
                                last_activation=last_layer_activation)
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dropout_7 (Dropout)          (None, 38654)             0         
_________________________________________________________________
dense_7 (Dense)              (None, 50)                1932750   
_________________________________________________________________
dropout_8 (Dropout)          (None, 50)                0         
_________________________________________________________________
dense_8 (Dense)              (None, 50)                2550      
_________________________________________________________________
dropout_9 (Dropout)          (None, 50)                0         
_________________________________________________________________
dense_9 (Dense)              (None, 3)                 153       
=================================================================
Total params: 1,935,453
Trainable params: 1,935,453
Non-trainable params: 0
_________________________________________________________________
None
```

In [94]:
```
# fit to tfidf vectorized
tfidf_model.fit(X_train_tfidf, y_train_encoded,
                epochs=10,
                batch_size=100,
                validation_split=0.2)
```

```
Train on 5348 samples, validate on 1337 samples
Epoch 1/10
5348/5348 [==============================] - 7s 1ms/step - loss: 0.9968 -
accuracy: 0.5869 - val_loss: 0.8614 - val_accuracy: 0.6028
Epoch 2/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.8337 -
accuracy: 0.6040 - val_loss: 0.8130 - val_accuracy: 0.6028
Epoch 3/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.7577 -
accuracy: 0.6322 - val_loss: 0.7771 - val_accuracy: 0.6200
Epoch 4/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.6529 -
accuracy: 0.7244 - val_loss: 0.7502 - val_accuracy: 0.6597
Epoch 5/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.5292 -
accuracy: 0.7999 - val_loss: 0.7721 - val_accuracy: 0.6552
Epoch 6/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.4280 -
accuracy: 0.8319 - val_loss: 0.8058 - val_accuracy: 0.6530
Epoch 7/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.3619 -
accuracy: 0.8687 - val_loss: 0.8365 - val_accuracy: 0.6402
Epoch 8/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.3124 -
accuracy: 0.8874 - val_loss: 0.8685 - val_accuracy: 0.6545
Epoch 9/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.2678 -
accuracy: 0.9041 - val_loss: 0.9082 - val_accuracy: 0.6507
Epoch 10/10
5348/5348 [==============================] - 6s 1ms/step - loss: 0.2630 -
accuracy: 0.9074 - val_loss: 0.9011 - val_accuracy: 0.6552
```

Out[94]:
```
<keras.callbacks.callbacks.History at 0x7f976245e4a8>
```

Similar to count vectorized data, overfitting starts occuring after epoch 3.

In [95]:
```
# build and compile with 3 epochs
count_model = build_ngram_model(input_shape=X_train_tfidf.shape[1:],
                                last_units=last_layer_units,
                                last_activation=last_layer_activation)

# train with 3 epochs
count_model.fit(X_train_tfidf,
                y_train_encoded,
                epochs=3,
                batch_size=100,
                validation_split=0.2)
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout_10 (Dropout)         (None, 38654)             0
_________________________________________________________________
dense_10 (Dense)             (None, 50)                1932750
_________________________________________________________________
dropout_11 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_11 (Dense)             (None, 50)                2550
_________________________________________________________________
dropout_12 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_12 (Dense)             (None, 3)                 153
=================================================================
Total params: 1,935,453
Trainable params: 1,935,453
Non-trainable params: 0
_________________________________________________________________


None

Train on 5348 samples, validate on 1337 samples
Epoch 1/3
5348/5348 [==============================] - 7s 1ms/step - loss: 1.0024 -
accuracy: 0.5890 - val_loss: 0.8776 - val_accuracy: 0.6028
Epoch 2/3
5348/5348 [==============================] - 6s 1ms/step - loss: 0.8389 -
accuracy: 0.6032 - val_loss: 0.8164 - val_accuracy: 0.6028
Epoch 3/3
5348/5348 [==============================] - 6s 1ms/step - loss: 0.7565 -
accuracy: 0.6414 - val_loss: 0.7768 - val_accuracy: 0.6343
```

Out[95]:
```
<keras.callbacks.callbacks.History at 0x7f979cc02f98>
```

In [96]:
```
_, nn_tfidf_train_score = tfidf_model.evaluate(X_train_tfidf, y_train_encod
_, nn_tfidf_test_score = tfidf_model.evaluate(X_test_tfidf, y_test_encoded,

print(f'Train Accuracy (Neural Network / TF-IDF Vectorized): {nn_tfidf_trai
print(f'Test Accuracy (Neural Network / TF-IDF Vectorized): {nn_tfidf_test_
```

```
Train Accuracy (Neural Network / TF-IDF Vectorized): 0.8952879309654236
Test Accuracy (Neural Network / TF-IDF Vectorized): 0.6590399146080017
```

Looking at the results on the TF-IDF vectorized dataset, we see similar testing set results, but more levels of overfitting present to the training data. Try on W2V data

N-Gram Neural Network (Word2Vec)

In [97]:
```
# build and compile
w2v_model = build_ngram_model(input_shape=X_train_w2v.shape[1:],
                              last_units=last_layer_units,
                              last_activation=last_layer_activation)
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout_13 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_13 (Dense)             (None, 50)                2550
_________________________________________________________________
dropout_14 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_14 (Dense)             (None, 50)                2550
_________________________________________________________________
dropout_15 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_15 (Dense)             (None, 3)                 153
=================================================================
Total params: 5,253
Trainable params: 5,253
Non-trainable params: 0
_________________________________________________________________
None
```

In [98]:
```
# fit to w2v vectorized
w2v_model.fit(X_train_w2v, y_train_encoded,
                epochs=100,
                batch_size=100,
                validation_split=0.2)
```

```
Epoch 76/100
5348/5348 [==============================] - 0s 53us/step - loss: 0.8086
- accuracy: 0.6083 - val_loss: 0.8045 - val_accuracy: 0.6081
Epoch 77/100
5348/5348 [==============================] - 0s 52us/step - loss: 0.8093
- accuracy: 0.6124 - val_loss: 0.8034 - val_accuracy: 0.6156
Epoch 78/100
5348/5348 [==============================] - 0s 52us/step - loss: 0.8046
- accuracy: 0.6094 - val_loss: 0.8055 - val_accuracy: 0.6118
Epoch 79/100
5348/5348 [==============================] - 0s 52us/step - loss: 0.8066
- accuracy: 0.6098 - val_loss: 0.8034 - val_accuracy: 0.6088
Epoch 80/100
5348/5348 [==============================] - 0s 52us/step - loss: 0.7981
- accuracy: 0.6182 - val_loss: 0.8025 - val_accuracy: 0.6148
Epoch 81/100
5348/5348 [==============================] - 0s 53us/step - loss: 0.8098
- accuracy: 0.6105 - val_loss: 0.8011 - val_accuracy: 0.6111
Epoch 82/100
5348/5348 [==============================] - 0s 55us/step - loss: 0.8067
```

```
word2vec vectorized is not overfitting as badly, but results are not as
strong as other neural nets run so far, with testing accuracy coming in
around ~60%.
```

In [99]:
```
_, nn_w2v_train_score = w2v_model.evaluate(X_train_w2v, y_train_encoded, ve
_, nn_w2v_test_score = w2v_model.evaluate(X_test_w2v, y_test_encoded, verbo

print(f'Train Accuracy (Neural Network / TF-IDF Vectorized): {nn_w2v_train_
print(f'Test Accuracy (Neural Network / TF-IDF Vectorized): {nn_w2v_test_sc
```

```
Train Accuracy (Neural Network / TF-IDF Vectorized): 0.6251308917999268
Test Accuracy (Neural Network / TF-IDF Vectorized): 0.624495267868042
```

Minimal levels of overfitting present and decently strong results with testing accuracy over 62%. May be possible to improve results with additional data or more epochs, despite already using a large number. Performance is under other methods identified.

5. Evaluation

Now that all models have been run, we can summarize results and run on full dataset.

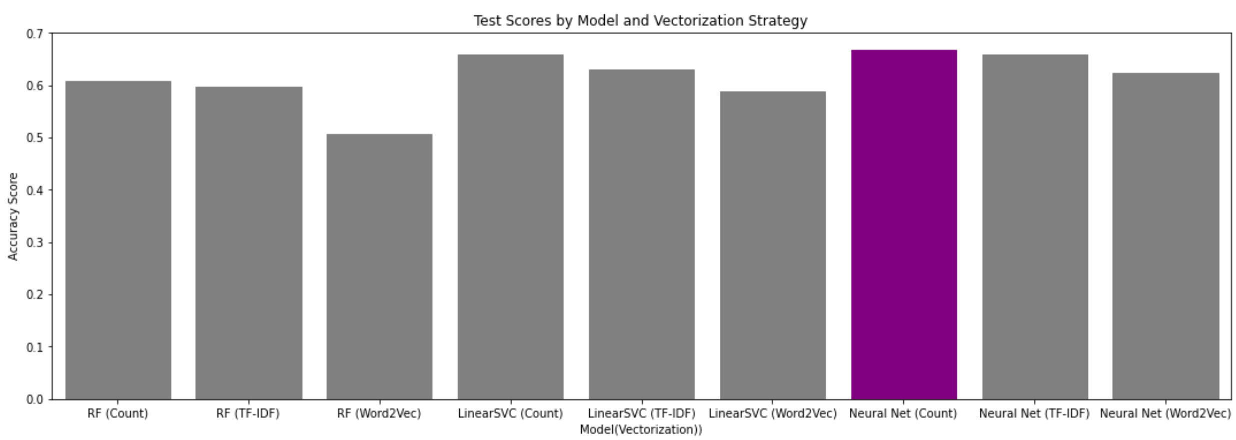
To start, summarize best identified scores from models run.

In [100]:
```
# summarize training and test scores for all models run:
best_rf_models = [best_rf_count, best_rf_tfidf, best_rf_w2v]
best_svc_models = [best_svc_count, best_svc_tfidf, best_svc_w2v]
print(f'Best Tuned Random Forest (Count Vectorized) Test Accuracy: {best_rf
print(f'Best Tuned Random Forest (Count Vectorized) Train Accuracy: {best_r
print(f'Best Tuned Random Forest (TF-IDF Vectorized) Test Accuracy: {best_r
print(f'Best Tuned Random Forest (TF-IDF Vectorized) Train Accuracy: {best_
print(f'Best Tuned Random Forest (Word2Vec) Test Accuracy: {best_rf_w2v.sco
print(f'Best Tuned Random Forest (Word2Vec) Train Accuracy: {best_rf_w2v.sc
print(f'Best Tuned LinearSVC (Count) Test Accuracy: {best_svc_count.score(X
print(f'Best Tuned LinearSVC (Count) Train Accuracy: {best_svc_count.score(
print(f'Best Tuned LinearSVC (TF-IDF) Test Accuracy: {best_svc_tfidf.score(
print(f'Best Tuned LinearSVC (TF-IDF) Train Accuracy: {best_svc_tfidf.score
print(f'Best Tuned LinearSVC (w2v) Test Accuracy: {best_svc_w2v.score(X_tes
print(f'Best Tuned LinearSVC (w2v) Train Accuracy: {best_svc_w2v.score(X_tr
print(f'Best Neural Network (Count) Test Accuracy: {nn_count_test_score}')
print(f'Best Neural Network (Count) Train Accuracy: {nn_count_train_score}'
print(f'Best Neural Network (TF-IDF) Test Accuracy: {nn_tfidf_test_score}')
print(f'Best Neural Network (TF-IDF) Train Accuracy: {nn_tfidf_train_score}
print(f'Best Neural Network (Word2Vec) Test Accuracy: {nn_w2v_test_score}')
print(f'Best Neural Network (Word2Vec) Train Accuracy: {nn_w2v_train_score}
```

```
Best Tuned Random Forest (Count Vectorized) Test Accuracy: 0.608344549125
1682
Best Tuned Random Forest (Count Vectorized) Train Accuracy: 0.68706058339
56619
Best Tuned Random Forest (TF-IDF Vectorized) Test Accuracy: 0.59712875729
02647
Best Tuned Random Forest (TF-IDF Vectorized) Train Accuracy: 0.7108451757
666417
Best Tuned Random Forest (Word2Vec) Test Accuracy: 0.5074024226110363
Best Tuned Random Forest (Word2Vec) Train Accuracy: 0.5968586387434555
Best Tuned LinearSVC (Count) Test Accuracy: 0.65814266487214
Best Tuned LinearSVC (Count) Train Accuracy: 0.7123410620792819
Best Tuned LinearSVC (TF-IDF) Test Accuracy: 0.6294302377747869
Best Tuned LinearSVC (TF-IDF) Train Accuracy: 0.630964846671653
Best Tuned LinearSVC (w2v) Test Accuracy: 0.5895020188425303
Best Tuned LinearSVC (w2v) Train Accuracy: 0.5992520568436799
Best Neural Network (Count) Test Accuracy: 0.6680125594139099
Best Neural Network (Count) Train Accuracy: 0.7880328893661499
Best Neural Network (TF-IDF) Test Accuracy: 0.6590399146080017
Best Neural Network (TF-IDF) Train Accuracy: 0.8952879309654236
Best Neural Network (Word2Vec) Test Accuracy: 0.624495267868042
Best Neural Network (Word2Vec) Train Accuracy: 0.6251308917999268
```

In [109]:
```
score_list = [best_rf_count.score(X_test_count, y_test),
              best_rf_tfidf.score(X_test_tfidf, y_test),
              best_rf_w2v.score(X_test_w2v, y_test),
              best_svc_count.score(X_test_count, y_test),
              best_svc_tfidf.score(X_test_tfidf, y_test),
              best_svc_w2v.score(X_test_w2v, y_test),
              nn_count_test_score,
              nn_tfidf_test_score,
              nn_w2v_test_score]
```

In [114]:
```
# visualize testing scores
test_labels = pd.Series(['RF (Count)',
                         'RF (TF-IDF)',
                         'RF (Word2Vec)',
                         'LinearSVC (Count)',
                         'LinearSVC (TF-IDF)',
                         'LinearSVC (Word2Vec)',
                         'Neural Net (Count)',
                         'Neural Net (TF-IDF)',
                         'Neural Net (Word2Vec)'])

test_acc = pd.Series(score_list)

plt.figure(figsize=(15, 5))
ax = sns.barplot(x=test_labels, y=test_acc, color='grey')
plt.tight_layout()
plt.title('Test Scores by Model and Vectorization Strategy')
plt.xlabel('Model(Vectorization))')
plt.ylabel('Accuracy Score')

# highlight the model/vectorization strategy with highest testing accuracy
max_height = max(score_list)
for bar in ax.patches:
    if bar.get_height() == max_height:
        bar.set_color('purple')
    else:
        bar.set_color('grey')

plt.show()
```



Looking at the comparison of accuracy scores between our different model, it appears using a neural network with count vectorized data is producing the best results. Additionally, when considering training times, LinearSVC performed the best, with Neural Networks coming second. Random Forest training times, including grid search seemed to take the longest, especially when including a large number of grid search params. Despite longer training time than LinearSVC, training time is not large enough to offset gains in performance.

Now run model on full training / test set with this model. Then run on full dataset.

In [119]:

```
# build model
best_model = build_ngram_model(input_shape=X_train_count.shape[1:],
                               last_units=last_layer_units,
                               last_activation=last_layer_activation)

# train with 3 epochs
best_model.fit(X_train_count, y_train_encoded,
               epochs=3,
               batch_size=100,
               validation_split=0.2)
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout_28 (Dropout)         (None, 38654)             0
_________________________________________________________________
dense_28 (Dense)             (None, 50)                1932750
_________________________________________________________________
dropout_29 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_29 (Dense)             (None, 50)                2550
_________________________________________________________________
dropout_30 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_30 (Dense)             (None, 3)                 153
=================================================================
Total params: 1,935,453
Trainable params: 1,935,453
Non-trainable params: 0
_________________________________________________________________

None

Train on 5348 samples, validate on 1337 samples
Epoch 1/3
5348/5348 [==============================] - 7s 1ms/step - loss: 1.0007 - accuracy: 0.5598 - val_loss: 0.8710 - val_accuracy: 0.6028
Epoch 2/3
5348/5348 [==============================] - 6s 1ms/step - loss: 0.8199 - accuracy: 0.6135 - val_loss: 0.7858 - val_accuracy: 0.6223
Epoch 3/3
5348/5348 [==============================] - 6s 1ms/step - loss: 0.7151 - accuracy: 0.6861 - val_loss: 0.7549 - val_accuracy: 0.6604
```

Out[119]:

```
<keras.callbacks.callbacks.History at 0x7f97be6c2f28>
```

In [122]:
```
_, best_train_score = best_model.evaluate(X_train_count, y_train_encoded, v
_, best_test_score = best_model.evaluate(X_test_count, y_test_encoded, verb

print(f'Train Accuracy (Neural Network / Count Vectorized): {best_train_sco
print(f'Test Accuracy (Neural Network / Count Vectorized): {best_test_score
```

```
Train Accuracy (Neural Network / Count Vectorized): 0.7774121165275574
Test Accuracy (Neural Network / Count Vectorized): 0.6720502376556396
```

Looking at the final best model selected, the neural network vectorized using count data, we see a testing score of 67%. Looking at the training score of ~77%, we see we are not overfitting that much to the training set. When considering a balanced set with simple model that could achieve ~33% accuracy with three classes to choose from, our model is vastly outperforming that. Also, when considering our specific case of imbalanced data, we know that the neutral class takes up roughly ~60% of the entire dataset. In that case, a simple model could achieve up to ~60% accuracy score. Even in this scenario, our best identified model is outperforming this 60%.

When considering these results in the real world, our classifier should correctly be able to identify whether a tweet is negative, positive, or neutral about 67% of the time. These results are somewhat strong, but given the initial class imbalance and lack of negative class entries, we would likely hope to achieve more data in the future to further improve results.

In conclusion, stakeholders using our classifier would be able to correctly classify ~67% of the tweets seen.

In [129]:
```
# save model using pickle
import pickle
with open('best.pickle', 'wb') as f:
    pickle.dump(best_model, f)
```

In [130]:
```
# load model
with open('best.pickle', 'rb') as file:
    best_model_2 = pickle.load(file)
```