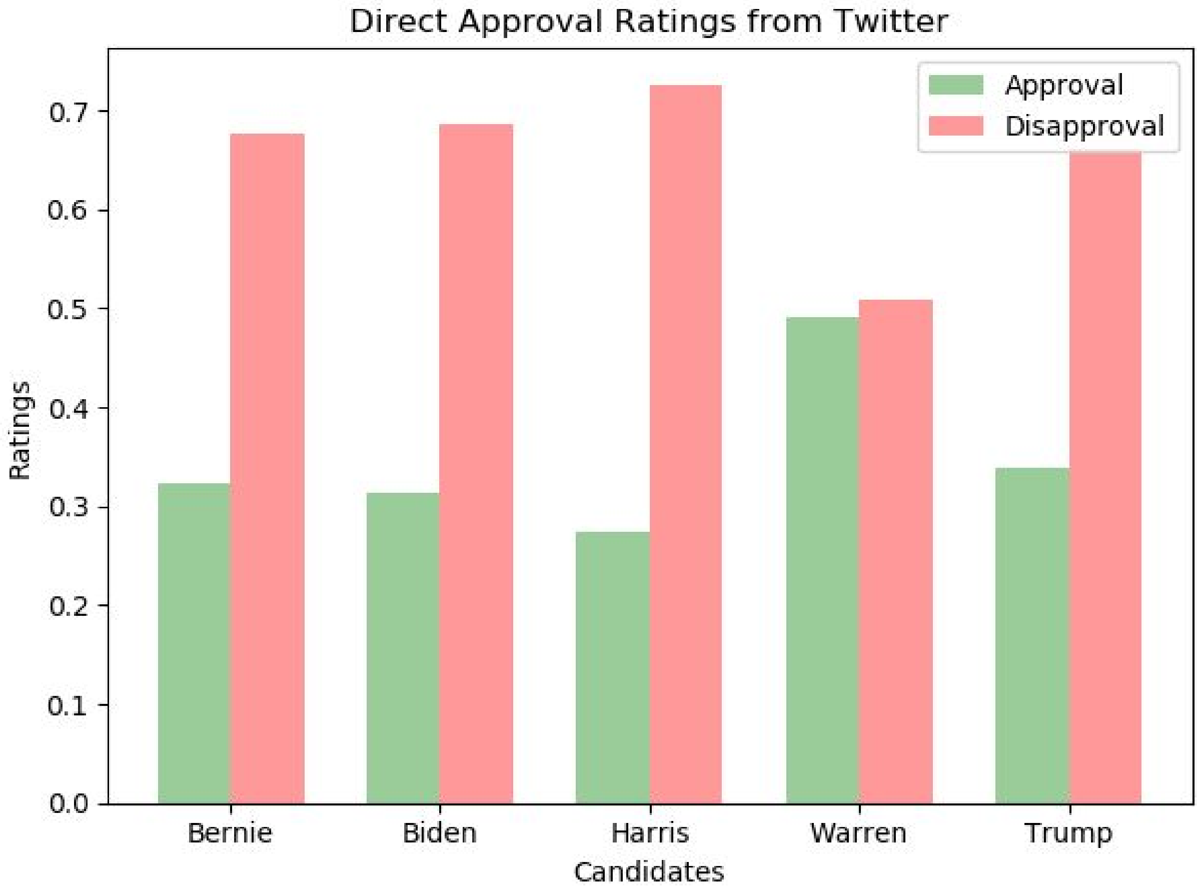
CIS 400: Principles of Social Media and Mining

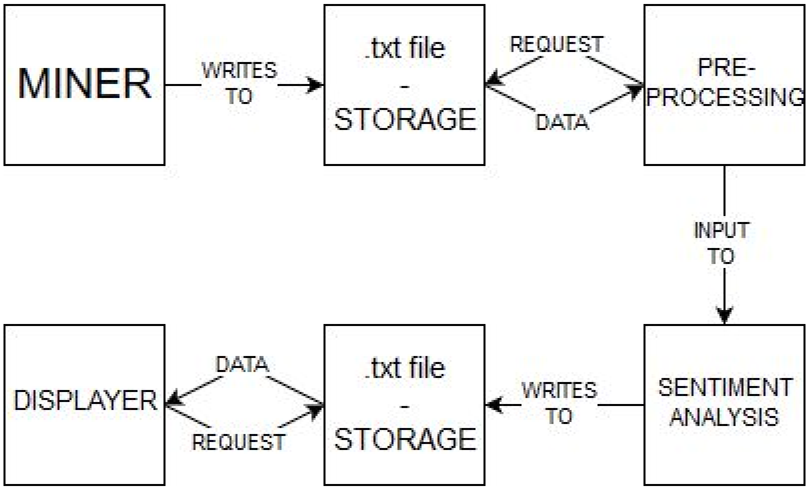
Predicting the Outcome of the 2020 Democratic Primary using Twitter

gitlab.com/ssants/CIS400-Final



Aiden Herlan, Timothy Kilmer, Sergio Santos, Alan Zumou, Jay Hoffacker

Approach



When brainstorming an idea for our final project, we considered many data mining applications that were both feasible and purposeful. However, the one that stuck out the most to us was the predicting the outcome of the Democratic and Republican Primaries of the 2020 presidential election. Eventually we found that doing an analysis on the Republican primary wouldn't be interesting because currently there are only two major candidates in the running and Donald Trump is the overwhelming favorite. Nevertheless, we wanted to develop our program so that it could carry out an analysis on any candidate of any party at any moment leading up to any election, and as such was designed with the intention of being universally applicable. We felt that the notion behind aiding voters in becoming more informed was an important problem that today's society has yet found a solution to. When determining the most optimal approach to this problem, we immediately gravitated towards creating a sentiment analysis that gauged user's attitude toward various candidates on one of the most widely used social media platforms, Twitter.

To do this, we acknowledged that we would first have to develop a miner to gather tweets that mentioned candidates partaking in the Democratic primary. For the miner, we determined that the best way to mine tweets was to leverage aspects unique to social media like hashtags and user mentions for each candidate. We would store various information regarding the user who authored the tweet like follower count, name, etc. Then we would store these mined tweets in a dictionary which was in turn stored into a text file corresponding to the candidate.

In addition to our sentiment analysis, we agreed that performing a centrality analysis to determine the influence of users we mined tweets from based on follower count would have a positive impact on the accuracy of our end results. To explain this in more detail, it's necessary to introduce a brief description of our sentiment analysis -- that will be further explained in more detail in a later section of this report. The goal of our sentiment analysis algorithm, in a nutshell, is to, with a trained naive bayes classifier and use it within a textblob object to spit out the sentiment value of a tweet after it's been cleaned and normalized. Afterwards, we calculate the number of positive and negative tweets per candidate and establish a positive to negative ratio to help us determine the most likely candidate to win the Democratic primaries. The thought behind implementing and incorporating a katz centrality into the results of our sentiment analysis was to weight our data in some meaningful and accurate way. Based on the results of our katz centrality analysis, performed on a network of twitter users for each candidate individually, we would establish a weight to the user of each tweet we collect, and apply it to the weight of each tweet in accordance i.e. a higher weight for more influential user with a higher following count and vice versa for less influential users. After weighting the data, we would then create a tally system to determine the approval ratings of each candidate.

Data/Approach cont.

<u>Mining for tweets on twitter</u>

In deciding what types of tweets to mine for our analysis, we came up with a set of keywords to establish which tweets would be most helpful in ensuring their intent, and thus, relevancy towards our project goals. An example of some of the keywords we used in filtering tweets include 'warren', ewarren', and 'elizabeth warren' for Elizabeth Warren, and 'bernie', and 'sanders' for Bernie Sanders.

Another important feature of our data mining program is accounting for retweets. In deciding the significance of retweets in the tabulation of both the positive and negative tweets of each candidate, we decided that twitter users who decided to retweet a tweet with a reasonably explicit political subject and/or reference to a political candidate were expressing a similar sentiment to that of the original tweet. As such, we decided to count retweets in an equal manner to that of the original. In separating retweets from original tweets -- something we needed to account for given a duplicate tweet removal feature in one of our data cleaning/normalizing routines, we added an identifier to the json string of each tweet object before we stored it in a file, and then compared the text attribute of each newly read in tweet object to see if there were any duplicate entries. Given the nature of our data collection procedures, utilization of the streaming API, and the separation of tasks for our system/sequence of events i.e. tabulation of results/sentiment analysis occurring after tweets were mined, we determined this to be the best way of accounting for retweets; as opposed to finding a way of using the retweet attribute of each tweet object.

Example Code for Data Collection

We utilize the tweepy library to collect a data set, or more specifically, a list of dictionaries for candidate. Given that size is an important determining factor in how accurate our results would turn out, we chose to make the size of each data set approximately 50,000 to 75,000 tweets.

We get our permission to access user accounts through OAuth authentication from the twitter developer website, as shown below.

```
auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
auth.set_access_token(OAUTH_TOKEN, OAUTH_TOKEN_SECRET)
```

We then query twitter for the tweets we needed to acquire for our project. So we use the names of the top five most-popular candidates (based on poll average data). For example:

```
track = ['Biden']
stream(track=track, limit=0, func=write_tweets)
```

While collecting the data, we extract only the valuable information from raw tweet, create a dictionary for each tweet, and store these dictionaries into local files.

```
            diction['followers'] = status.user.followers_count
            diction['created_at'] = str(status.created_at)
            diction['coords'] = status.coordinates
            diction['retweet_count'] = status.retweet_count
            diction['verified'] = ((status._json)['user'])['verified']
            try:
                diction['text'] =
((status._json)['extended_tweet'])['full_text']
…
                f = open(self.filename,"a")
            try:
                f.write(str(diction) + "\n")
…
                 f = open("trump.txt","a")
                f.write(str(coord) + "\n")
                f.close()
```

It’s important to note that we store the data set for each candidate we’re analyzing so that we don’t have to spend hours running our data mining application to collect new data sets for each candidate every time we want to test our system. Should there ever be a need to acquire and test a more recent set of data for each candidate, one need only run our data mining procedures to collect more data. Furthermore, storing each candidate in its own respective file lets us categorize each set of tweets, further simplifying the entire process.

To take the data we store in files and convert it into a list of dictionaries for our text processing and sentiment analysis routines, we the below function getAllTweets:

```
def getAllTweets(filename):
    """Returns all of the tweets in 'filename' as an
    array of dictionaries"""
    arrayOfTweets = []
```

```
        # Opens file for reading
        f = open(filename,"r")
        # Loops through each line in f
        for _, line in enumerate(f):
            # Parses the string representation of the dictionary to an actual
dictionary
            dictionary = strToDict(line)
            # Checks if the dictionary is not None,
            # If it is, than an error has occured, so don't add it to the list
            if dictionary is not None:
                # If it is not, than its a valid tweet, so add it to the list
                arrayOfTweets.append(dictionary)
        f.close()
        return arrayOfTweets
```

Once we have all the datasets we require in order to perform the sentiment analysis and have successfully stored them in our preprocessing/analysis code file, we then extract each tweet message from the tweet dictionary. This is accomplished by iterating through each dictionary within the data set and extracting -- through indexing the 'Text' key of each dictionary -- the tweet as a string; thus resulting in the construction of a list of tweets -- as strings.

After obtaining the dataset of tweets per each candidate we're analyzing, we run each one through a set of text preprocessing procedures in order to rid them of any content that isn't useful and/or may serve to lessen the accuracy of naives bayes classifier before running our sentiment analysis algorithm. Our process can be viewed below:

First, we remove duplicate tweets from data sets (having already accounted for retweets), after which we remove duplicate words from the tweets themselves. This is done in the *tokenize_str* function. This procedure is also where the tweets are tokenized into lists of words, and where special characters are removed -- while contractions are maintained -- which proves useful in performing the following set of routines. For duplicate handling, there was an argument to be made that the remove duplicates function was completely unnecessary, and that simply accounting for all duplicates as tweets to be equally counted would of been an easier approach, but we felt it necessary to for the purpose of maximizing the accuracy of our findings that accounting for an instance in which a twitter user tweeted something twice -- for any reason -- should be accounted for.

```
twt_lst = tokenize_str(x) #tokenize tweet (account for contractions)
```

For tokenization, we import TweetTokenizer.

```
from nltk.tokenize import TweetTokenizer
def tokenize_str(str):
      tokenizer = TweetTokenizer()
```

```
tokenized_strList = tokenizer.tokenize(str)
```

We also use the .str() function.

The next step in the process was to remove stopwords from the tokenized tweet, given that they contribute nothing to the context of the tweet. This includes articles, transition words, etc.

```
twt_lst = remove_sw(twt_lst)           #remove stop words from tweet
```

For removing stopwords , we import a stopwords dictionary, and select the scope to be ‘ english’.

```
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
[tokenized_list.append(x) for x in lst if x not in stop_words]
```

For Lemmatization (takes place under normalization of data), we import wordnet, WordNetLemmatizer and pos_tag. To begin this section, it’s important to note that in order to successfully lemmatize a word, we must first know the type of the word; identified by a ‘tag.’ Pos_tag will provide tag of the word type of each word in the tokenized tweet. However, this tag set isn’t supported by the lemmatizer function’s tag set, thus we had to convert each tag to the appropriate WordNet tag so it could be incorporated into the lemmatizer’s tag set; be identified as the proper word type to which this word belongs i.e. adjective, verb, noun, etc. We use a helper function (taken from a source that will be cited in the code) that will convert an nltk tag to a wordnet tag. (i.e. wordnet.ADJ) At the end of this routine, the wordnet tag of the word and the word itself will be passed to the lemmatizer to produce the word’s lemmatized form.

```
lemmatizer = WordNetLemmatizer()
lst_pos = pos_tag(lst)
#create list of tuples (word, nltk pos)
      new_list = []
      for x in lst_pos:

      pos_t = pos_tag_finder(x[1])
      if (pos_t == ''):
             continue
      w = lemmatizer.lemmatize(x[0], pos_tag_finder(x[1]))
      if (w != ''):
            new_list.append(w)
            lst = new_list
      return lst
```

Example output:

```
#['fantasized', 'going', 'rocks', 'become'] -> ['fantasize',
'go', 'rock', 'become']
```

We then perform additional text normalization routines including case normalization and further tokenization routines including more targeted removal of special characters. Once Preprocessing is finished and we have a clean text (joined back into a string from a list of words), we begin our sentiment analysis/classification of tweets.

For classification, we import twitter_samples , TextBlob, NaiveBayesClassifier, and classify. To train our naive bayes classifier, we use samples from twitter_sample, which are already categorized into positive and negative tweets. We use 1000 of these sample tweets to train our NaiveBayesClassifier, as shown below. This gives us a classifier accuracy ranging from 75 to 78%. (commented code in the analysis program that outputs classifier accuracy). It is then combined into a textblob object, as shown below, after which the sentiment analysis is performed. We determine sentiment both from the sentiment function of textblob, and the classifier tags of ‘pos’ and ‘neg’ -- the ladder of which we’ve found to be significantly more beneficial in the accuracy of its findings i.e. in in determining whether a tweet is positive or negative.

```
cleaned_str = TextBlob(twt_lst,classifier=classifier)
sentimnt = cleaned_str.sentiment s_tuple = (("tweet:
"+str(counter)),sentimnt[0],cleaned_str.classify())
```

As mentioned above, to determine the influence of a tweet, we decided to look at the influence of the author by doing a centrality analysis on their immediate network. Instead of using degree centrality (which amounts to a popularity ranking) we first decided to use eigenvector centrality. To model the network, we used a directed network to preserve the flow of influence: node A connects to node B only if A follows B, and just because A connects to B doesn’t mean that B connects to A. Because the model is a directed graph, eigen centrality would return 0 to many nodes, even if they had over one million followers. To fix this, we switched to Katz centrality and assign $\boldsymbol{\beta}$ (beta) according to a scaled version of their follower count. Since the Katz centrality score would eventually be used as a weight during sentiment analysis, the final score shouldn’t be zero so we added one point to each score in case it was zero. In the end, the Katz centrality algorithm we used was:

$$C_{Katz}(v_i) = \alpha \sum_{j=1}^{n} A_{j,i} C_{Katz}(v_i) + \beta + 1$$

$$\text{where } \alpha = 0.85 \text{ and } \beta = \frac{user.follow-count}{10{,}000{,}000}$$

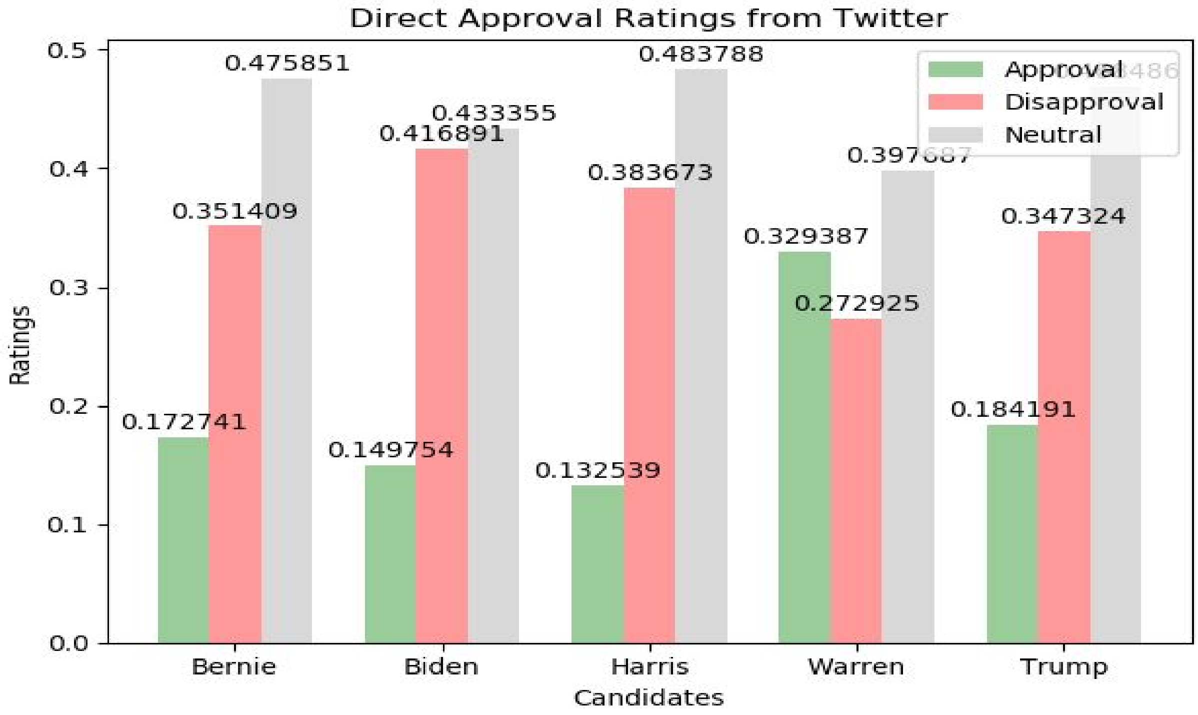
To implement this, the idea was to determine Katz centrality on a mini-network created by getting any significant followers (for example, one million or more followers) of the user in question. For the large majority of users we tested, the centrality score came out to 1 because they did not have any significant followers.

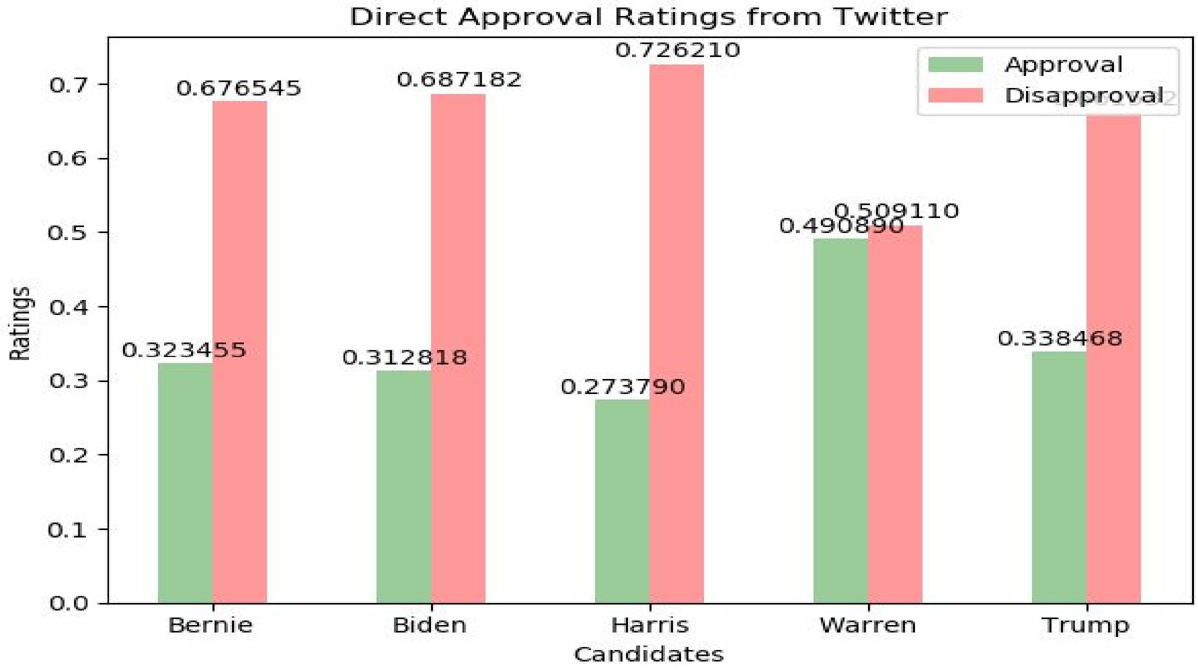
Centrality was a major topic of discussion in our program, and the program that was developed in order to implement and utilize a centrality analysis will be included in the submission of this project. However, in the end, we came to the conclusion that follower count wasn’t a sufficient means of determining a weight for our results. We also came to the conclusion that determining a proper set of attributes; attributes that directly correlate to an increase or decrease in the influence of overall sentiment to an individual or, in this case, a presidential candidate, had too large a scope. With that being said, the katz centrality analysis our group developed for follower count is significant enough to where we felt it necessary to discuss and include in this report. However, it wasn’t utilized in our end results.

Experimental Results

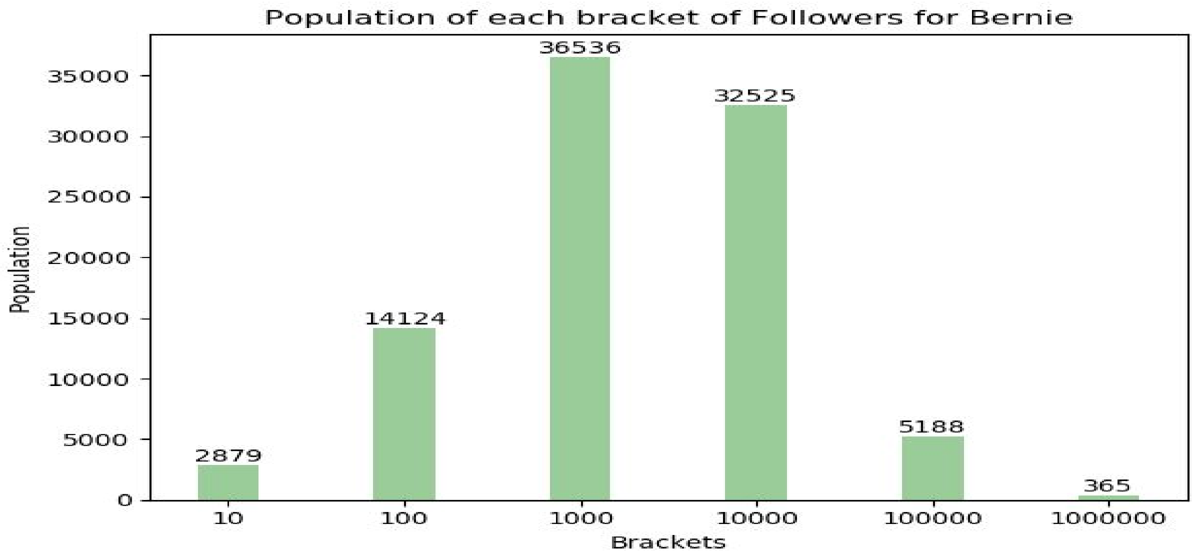
An important factor to make note of when viewing the graphs below is our usage of two separate utilities to compute sentiment, which we feel necessary to mention once more. Both the text blob’s sentiment.classifier functions and the naive bayes classifier, which after trained, was incorporated into the textblob object, work synchronously, however we found that in certain instances in which the sentiment function would produce an invalid output, the sentiment tag produced by the classifier alone gave an accurate sentiment reading of the tweet, which was taken into consideration when constructing the visual representations of our graphs.

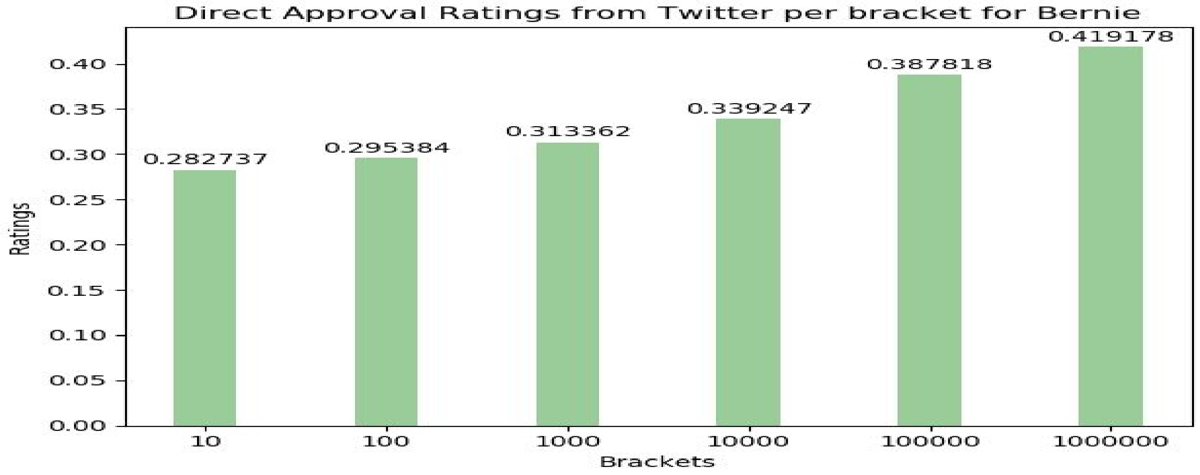
We came to a consensus that we wanted to use the library provided by MatPlot that Professor Yu mentioned in his lecture slides to display our data. Graphical representations of our results, alongside descriptions of each graph can be viewed below.

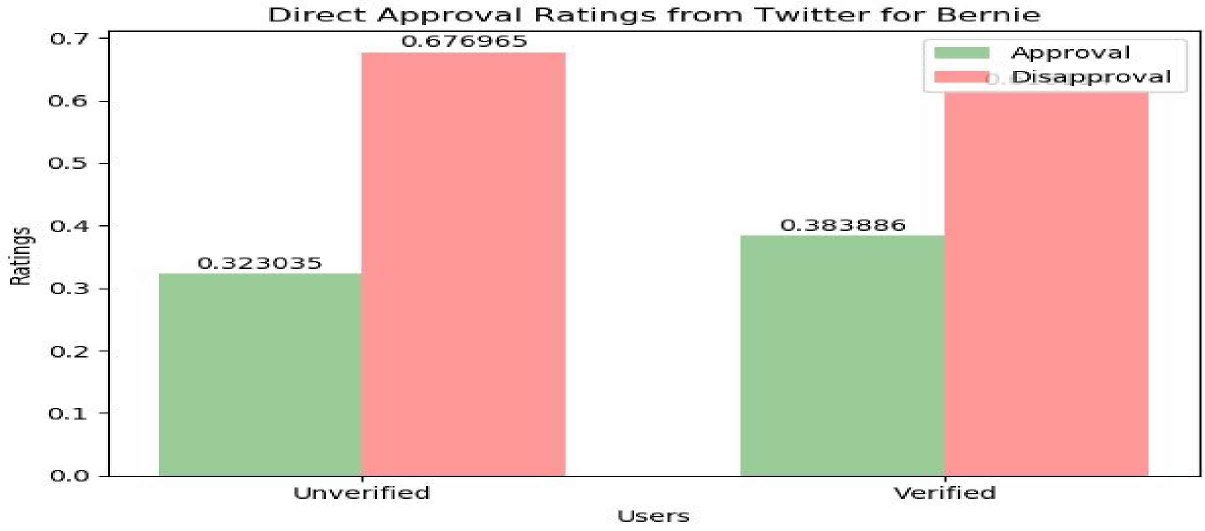

Direct Approval Ratings from Twitter
Approval
Disapproval
Neutral
Ratings
0.0
0.1
0.2
0.3
0.4
0.5
0.172741
0.351409
0.475851
0.149754
0.416891
0.433355
0.132539
0.383673
0.483788
0.329387
0.272925
0.397687
0.184191
0.347324
Bernie
Biden
Harris
Warren
Trump
Candidates


Direct Approval Ratings from Twitter
Approval
Disapproval
Ratings
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.323455
0.676545
0.312818
0.687182
0.273790
0.726210
0.490890
0.509110
0.338468
Bernie
Biden
Harris
Warren
Trump
Candidates

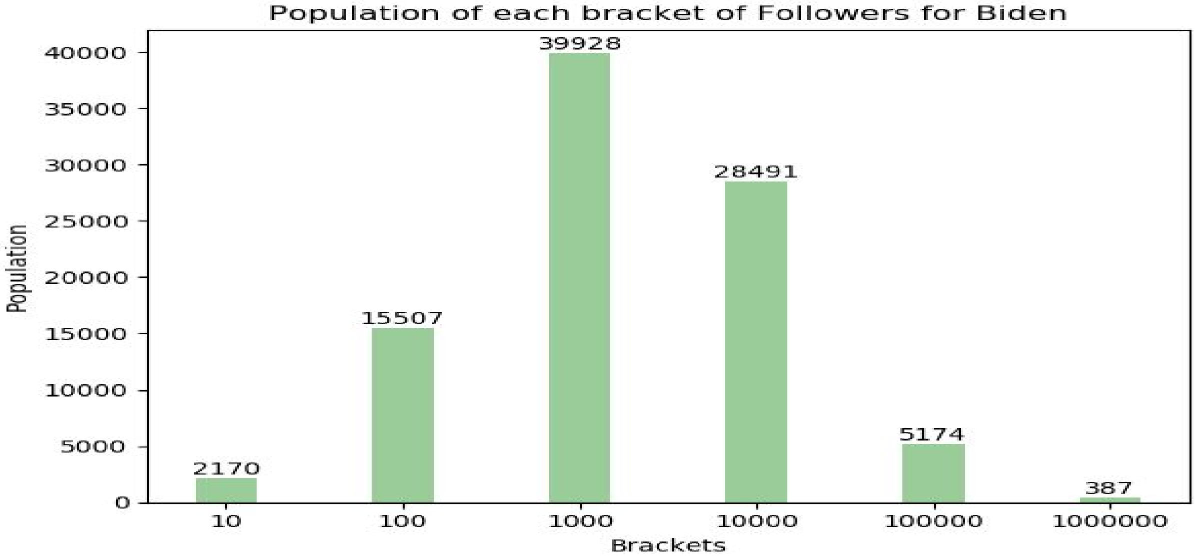
Bernie

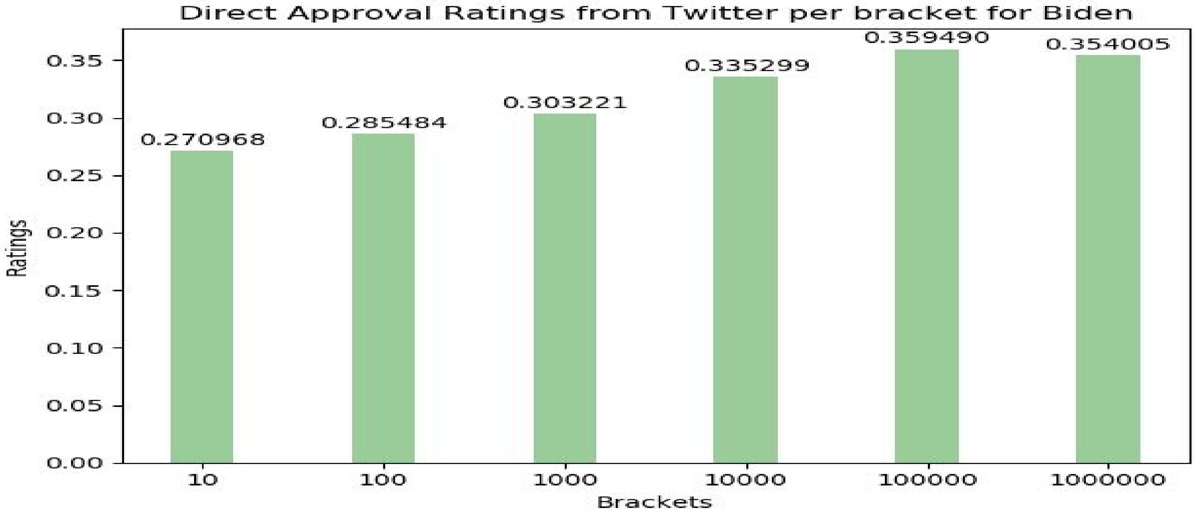

Population of each bracket of Followers for Bernie
Population
Brackets
0
5000
10000
15000
20000
25000
30000
35000
10
100
1000
10000
100000
1000000
2879
14124
36536
32525
5188
365

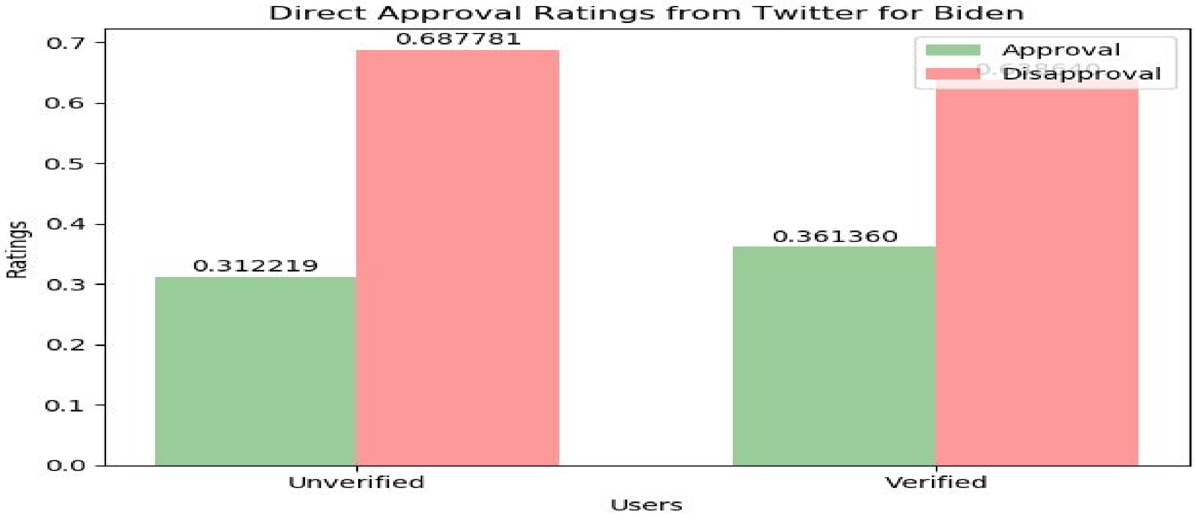

Direct Approval Ratings from Twitter per bracket for Bernie
Ratings
Brackets
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
10
100
1000
10000
100000
1000000
0.282737
0.295384
0.313362
0.339247
0.387818
0.419178


Direct Approval Ratings from Twitter for Bernie
Approval
Disapproval
Ratings
Users
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
Unverified
Verified
0.323035
0.676965
0.383886

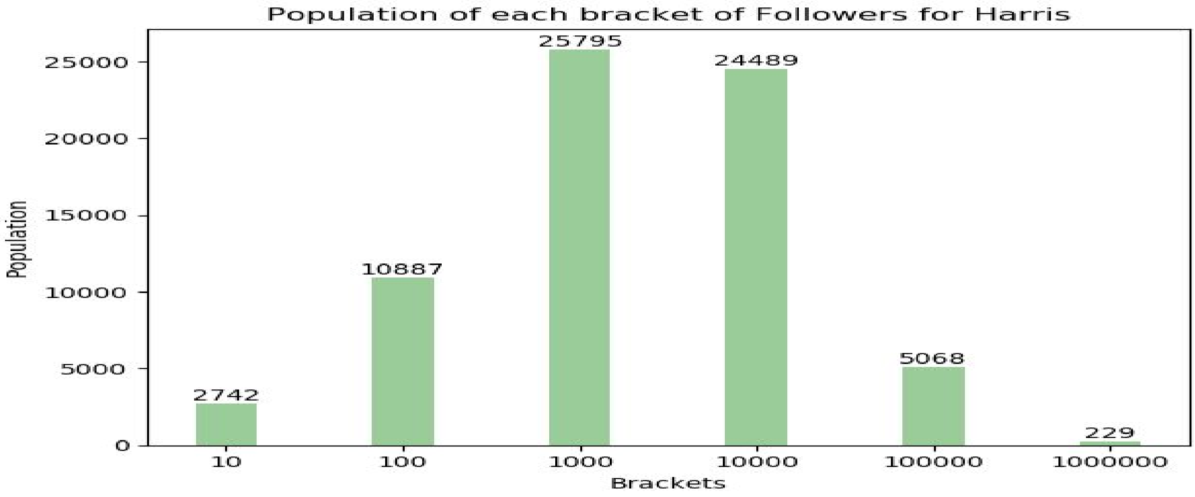
Biden

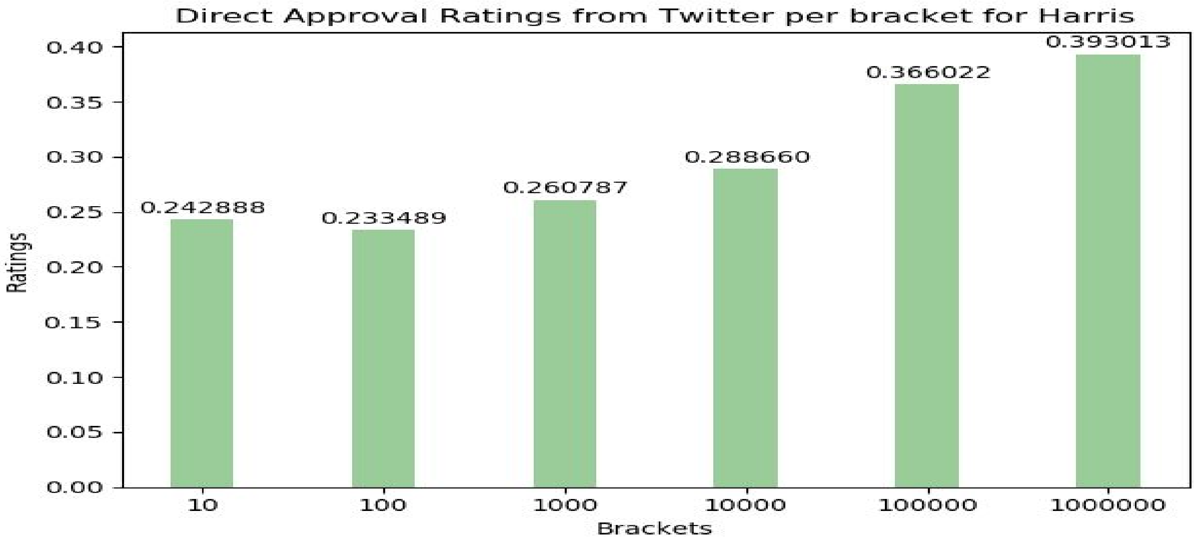

Population of each bracket of Followers for Biden
Population
40000
35000
30000
25000
20000
15000
10000
5000
0
2170
15507
39928
28491
5174
387
10
100
1000
10000
100000
1000000
Brackets

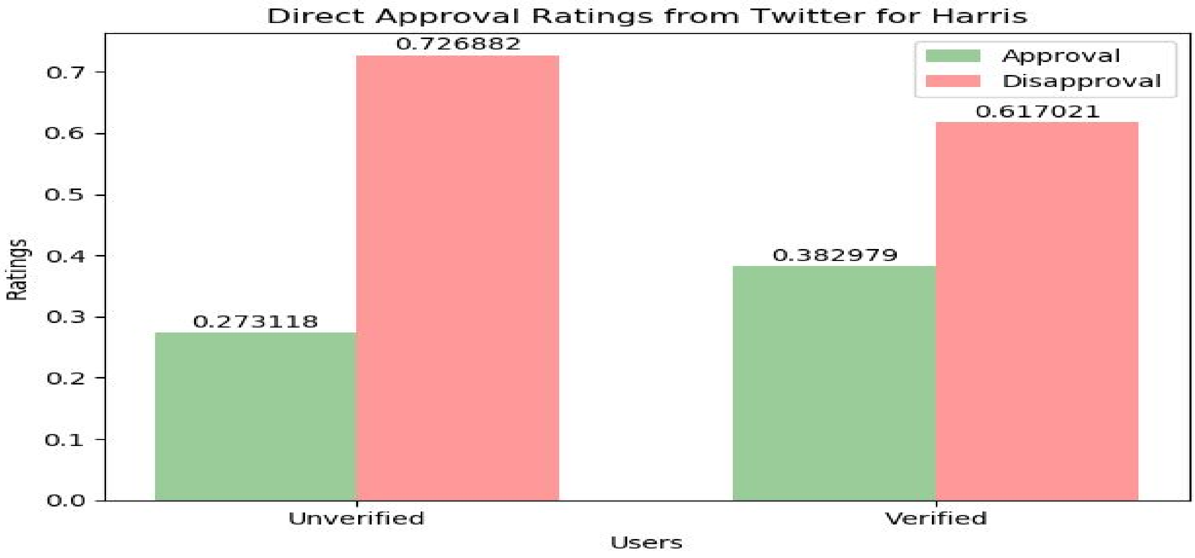

Direct Approval Ratings from Twitter per bracket for Biden
Ratings
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.270968
0.285484
0.303221
0.335299
0.359490
0.354005
10
100
1000
10000
100000
1000000
Brackets


Direct Approval Ratings from Twitter for Biden
Ratings
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Approval
Disapproval
0.312219
0.687781
0.361360
0.638640
Unverified
Verified
Users

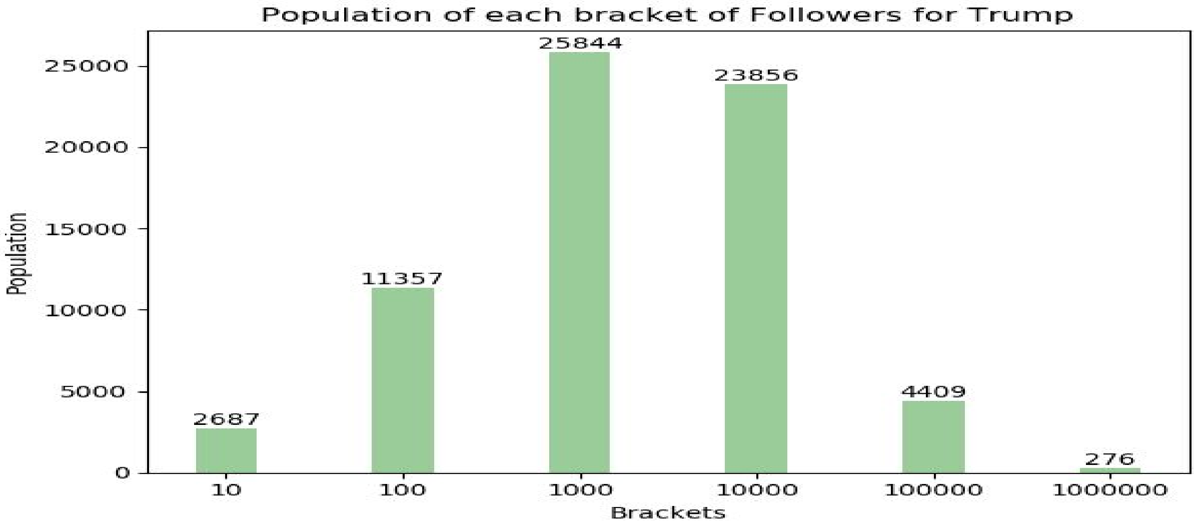
Harris

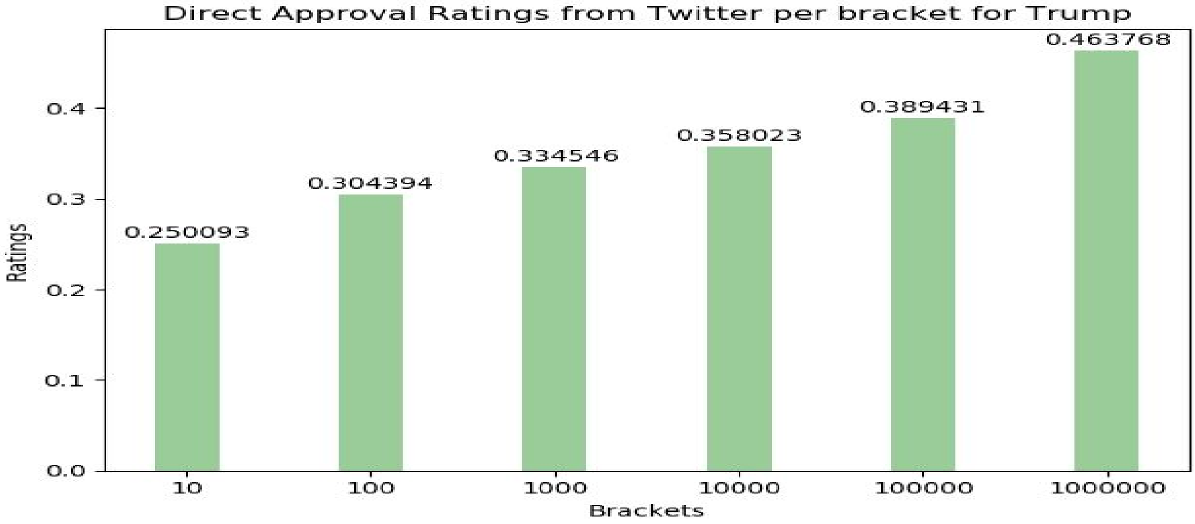

Population of each bracket of Followers for Harris
Population
0
5000
10000
15000
20000
25000
2742
10887
25795
24489
5068
229
10
100
1000
10000
100000
1000000
Brackets

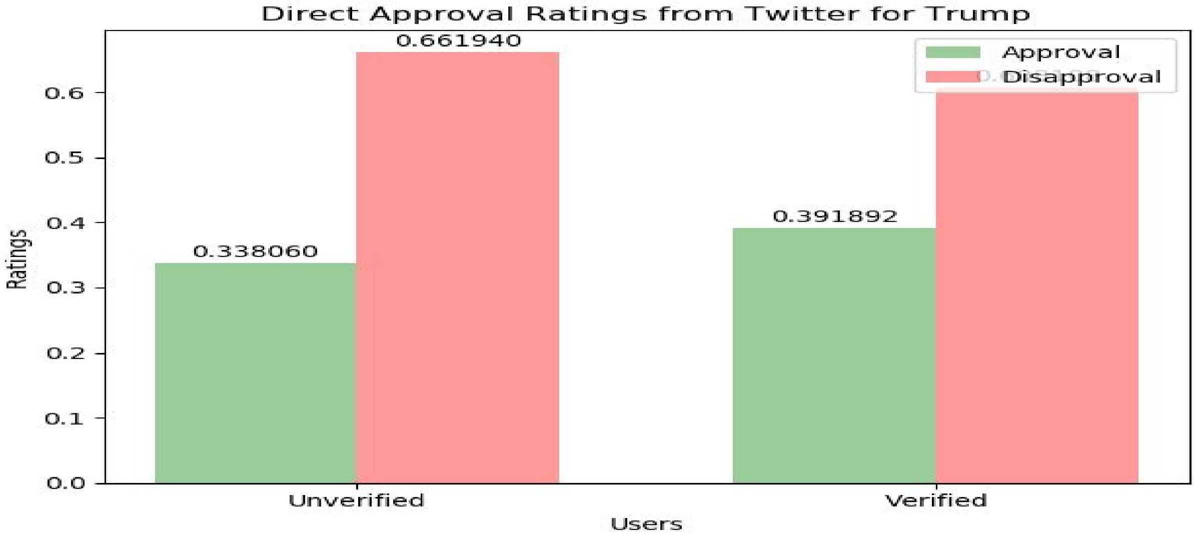

Direct Approval Ratings from Twitter per bracket for Harris
Ratings
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.242888
0.233489
0.260787
0.288660
0.366022
0.393013
10
100
1000
10000
100000
1000000
Brackets


Direct Approval Ratings from Twitter for Harris
Approval
Disapproval
Ratings
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.273118
0.726882
0.382979
0.617021
Unverified
Verified
Users

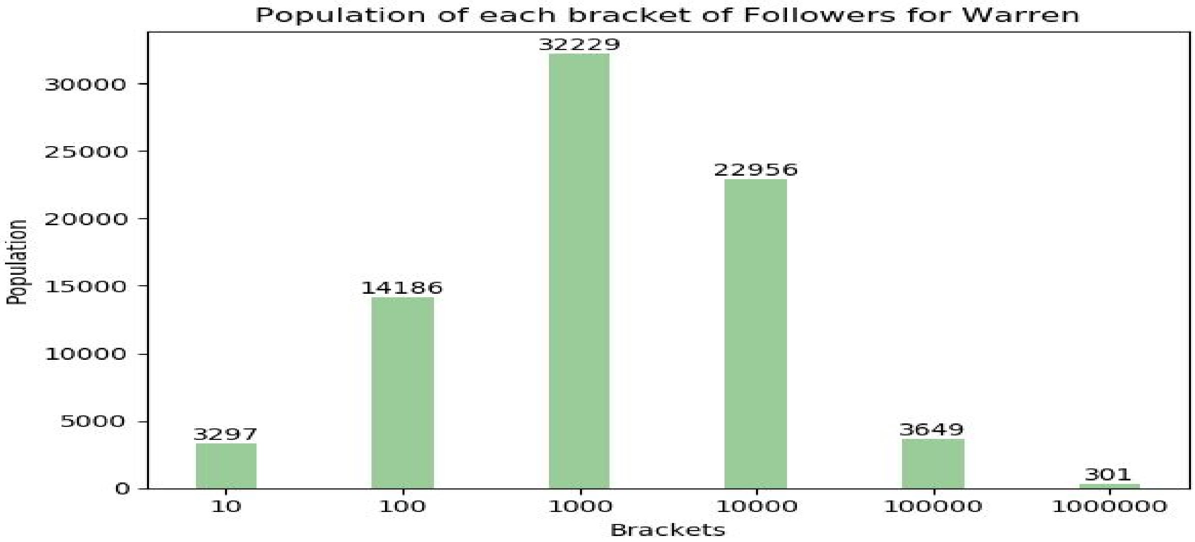
Trump

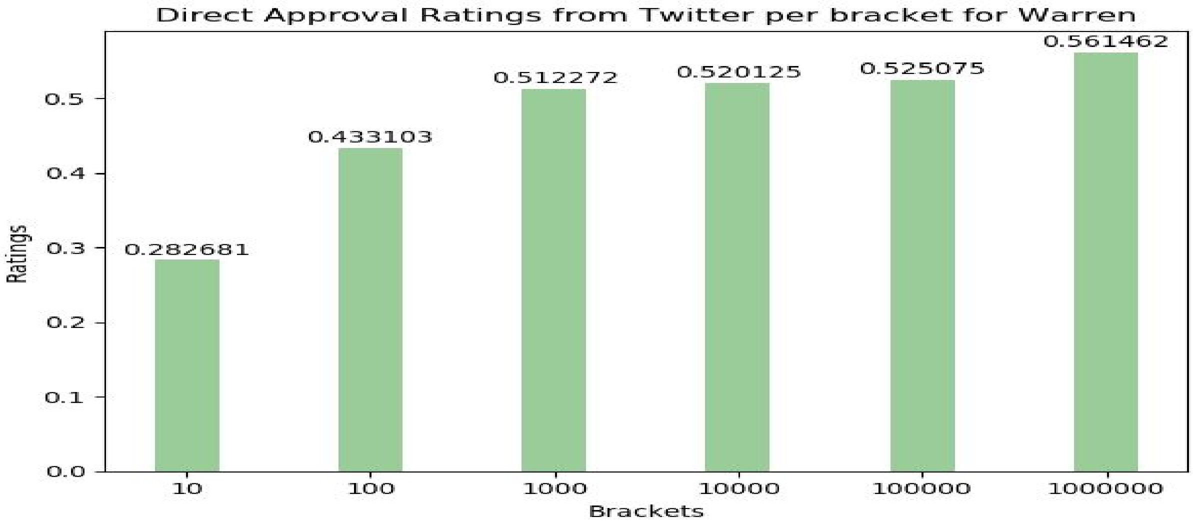

Population of each bracket of Followers for Trump
Population
25000
20000
15000
10000
5000
0
2687
11357
25844
23856
4409
276
10
100
1000
10000
100000
1000000
Brackets

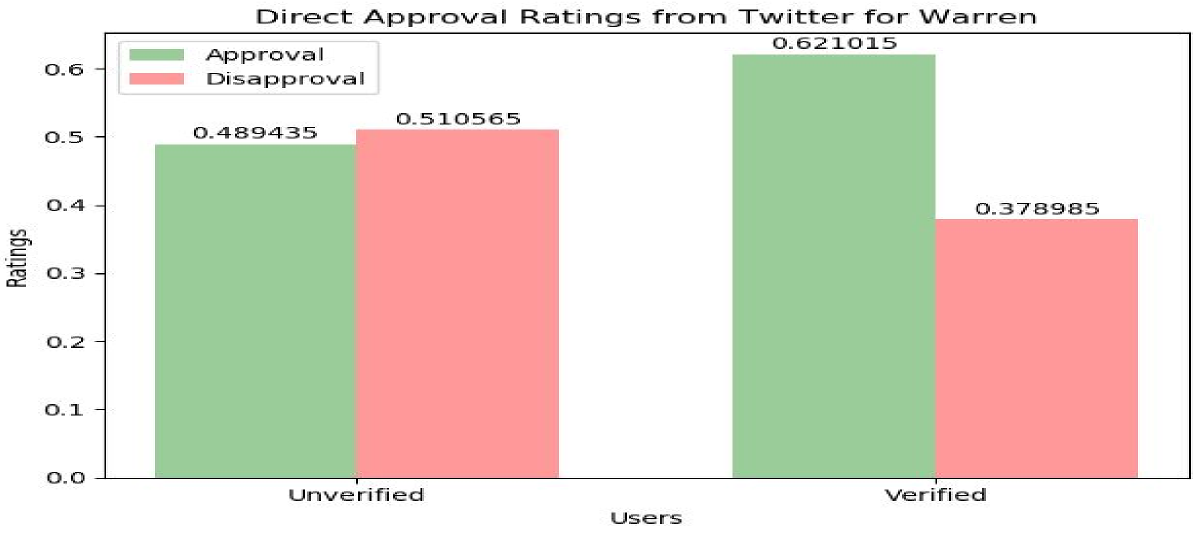

Direct Approval Ratings from Twitter per bracket for Trump
Ratings
0.4
0.3
0.2
0.1
0.0
0.250093
0.304394
0.334546
0.358023
0.389431
0.463768
10
100
1000
10000
100000
1000000
Brackets


Direct Approval Ratings from Twitter for Trump
Approval
Disapproval
Ratings
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.661940
0.338060
0.391892
Unverified
Verified
Users

Warren


Population of each bracket of Followers for Warren
Population
Brackets
0
5000
10000
15000
20000
25000
30000
10
100
1000
10000
100000
1000000
3297
14186
32229
22956
3649
301


Direct Approval Ratings from Twitter per bracket for Warren
Ratings
Brackets
0.0
0.1
0.2
0.3
0.4
0.5
10
100
1000
10000
100000
1000000
0.282681
0.433103
0.512272
0.520125
0.525075
0.561462


Direct Approval Ratings from Twitter for Warren
🟩 Approval
🟥 Disapproval
Ratings
Users
0.0
0.1
0.2
0.3
0.4
0.5
0.6
Unverified
Verified
0.489435
0.510565
0.621015
0.378985

As you can see, the distribution among the population sorted by follower count is normal, which makes sense. A clear trend among most of the candidates is that follower count is proportional to approval rating, especially in Warren's case. Most of the candidate's approval ratings separated by verified and unverified users show no significant difference. The exception is Warren, where the difference in approval rating among unverified users and verified users differ by ~14%. This may be due to many famous people approving of Warren's policies, but unless more intense analysis is done, it's unclear why there is such a large margin for Warren among verified and unverified users. For our aggregated results, we make one analysis where we considered tweets of any polarity, and one analysis where we said any tweet with a polarity of less than .1 or more than -.1 was a "neutral" response to the candidate in question. As we can see, most of the tweets about candidates were neutral, but the distribution of sentiment in the neural tweets is fairly uniform, as neural tweets were about half of all tweets for most candidates.

Conclusion

Social media plays an immense role in how voters view politicians. Most voters learn about a specific candidate through what they see on social media. In the age where voters get most if not all of their information about recent and important events, we found it useful to do our research based around user tweets accessed using Twitter's API. In doing so, we were able to conduct our research on tens of thousands of tweets to get a consensus on who is most likely to win the Democratic primaries for the upcoming 2020 election. The immense platform Twitter and other social media platforms present to data mining is remarkable and the possibilities are endless. Even though we mined data concerning the Democratic primaries, there are endless research opportunities presented by Twitter's API . Not only did our project present some insightful results based on the experiments we conducted, but can be expanded upon and modified for even more insight to the 2020 election and elections to come.