SURVEY OF POLITICAL BOTS ON TWITTER

By

DAVID TROUPE

A thesis submitted to the

Graduate School-Camden

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of Master of Science

Graduate Program in Scientific Computing

Written under the direction of

Dr. Jean-Camille Birget

And approved by

_____

Dr. Jean-Camille Birget

_____

Dr. Sunil Shende

_____

Dr. Suneeta Ramaswami

Camden, New Jersey

December 2018

THESIS ABSTRACT

Survey of Political Bots on Twitter

By: DAVID TROUPE

Thesis Director:

Dr. Jean-Camille Birget

Bots are software applications that execute automated tasks called scripts over the Internet. Bots have become predominant on social media platforms like Twitter, and automate their interactions with other users. Political Twitter bots have emerged that focus their activity on elections, policy issues, and political crises. These political bots have faced increased scrutiny as a result of their association with online manipulation via the spread of misinformation. As bots have become more sophisticated, research has focused on advanced methods to detect their presence on social media platforms. However, little research has been performed on the overall presence of political bots and their dynamic response to political events. The research that has been performed on political bots focuses on these bots in the context of scheduled political events, such as elections. In this paper, we explore the bot response to an unexpected political event, describe the overall presence of political bots on Twitter, and design and employ a model to identify them based on their user profile alone. We collected data for more than 700,00 accounts tweeting with hashtags related to political events in the United States between May 2018 and October 2018. We designed a machine learning algorithm using user

profile features alone that achieves approximately 97.4% accuracy in identifying political Twitter bots. In our analysis, we found (1) new bot accounts are created in response to political events, (2) bot accounts are more active during political controversies, (3) the number of tweets an account has favorited (liked) is a strong determinant of bot status.

Table of Contents

1 Understanding Twitter Bots

1.1 What is a Twitter bot?

An Internet bot, also known as web robot (or simply, bot) is a software application that runs automated tasks called scripts over the Internet[1]. Typically, bots perform simple and repetitive tasks at a rate much faster than humans are capable of. Bots are most commonly used for web-crawling and account for more than half of all web traffic [2]. Twitter bots are Internet bots that operate Twitter accounts, and automate their actions with other Twitter users [3].

However, the term “bot” is commonly used not only to refer to fully automated accounts, but also to describe “fake” Twitter accounts [4]. Twitter bots have been successfully used to promote businesses, provide customer service, and aid political campaigns [5]. Bots that abide by Twitter’s rules generate large amounts of harmless content, such as providing real-time news or weather updates. Malicious bots spread spam and other malicious content, such as fake news and misinformation [3]. Irrespective of their purpose, bots share the properties of being able to send messages and replicate themselves [6]. Twitter does not actually refer to these accounts as bots - instead, it refers to them as “Applications”. As of July 2018, all new Twitter applications must be approved via an application process. Despite this new regulation, applications existing prior to July of 2018 are not yet required to retroactively complete the application process [7].

1.2 What is a Twitter cyborg?

Cyborg is a term that refers to either human-assisted bots or bot-assisted humans. That implies that Cyborgs possess characteristics of both human and automated behavior [7]. In other words, cyborgs exist in the gray area between a fully automated account and a fully human-operated account. Cyborgs are common on Twitter. After registering an application with Twitter, a human will often automate some portion of the account's behavior, such as responding to an RSS feed or tweeting in the absence of the user. However, the user may still use this account to interact with other users. Cyborgs have also been described as sophisticated bots because of the difficulty associated with identifying them [8]. Just as with bots, their actions can be benign or malicious.

1.3 What is a political Twitter bot?

Political Twitter bots are accounts that focus their activity on elections, policy issues and political crises [9]. As bots, they may refer to fully automated accounts or accounts that users deem as "fake" - however, they may also share some traits of a cyborg. These bots have been used by politicians, government agencies, and advocacy groups in order to spread their desired message. Automated accounts have been caught disseminating lies, attacking people, and disrupting political conversations by assuming extreme political views [10][11]. Although political bots only gained notoriety after the US presidential election in 2016, they have been used extensively in the past. Political bots have been used in Congressional elections and Senate elections dating back to at least 2010 [9]. They have also been used to influence public opinion on politics outside of the US - for

example, by taking a stance on the Brexit debate about the UK's role in the EU [9] and promoting radical political ideologies in Venezuela [12]. Twitter has historically been very slow to act on the negative actions of these automated accounts [13]. This year, however, Twitter has been more actively trying to combat the problem of political bots on its platform [14]. Twitter has been engaging in extensive analysis into automated, election-related activity originating out of Russia on its platform, and has identified more than 50,000 political bots that were active during the 2016 US Presidential election [15]. Political bots pose a clear threat to democracies around the globe.

1.4 Challenges in identifying political bots

Identifying political Twitter bots can be difficult because aside from obvious flags (such as an account only retweeting other users), there is no simple set of rules by which an account can be identified as a bot [8]. Research has delineated some criteria that helps to identify bots, such as a lack of intelligent or original content, excessive automation or tweeting, abundant use of spam or malicious URLs, posting duplicate tweets, and mass following and unfollowing of accounts [5]. However, these criteria do not allow for the identification of all political bot-related activity. For instance, cyborgs (which may also generate political content and are often included under the umbrella of "political bots") exhibit both humanoid and automated characteristics. Thus, by solely examining the aforementioned criteria (which are geared more towards detecting automated activity), it still may be difficult to identify political bots.

Another challenge in identifying *political* bots in particular revolves around their targeted activity. A bot focusing on spam may be easily identified, whereas a bot focused on political activity may be harder to distinguish because its activity more closely resembles that of a human. Bots in general, and especially political bots, are often designed to give the impression they represent real people, providing authentic support for an issue. This may explain why, in general, it appears that it is more difficult for humans to identify bots than human accounts [8]. A great deal of effort is often used when creating fake personas for these accounts, thus making it difficult to distinguish these accounts from those of humans.

Some research has made use of a human turing test, which utilizes a human tester to identify "intelligence," an inherently human characteristic. The standard Turing tester communicates with an unknown subject for a fixed period of time, and decides whether it is a human or a machine. The Turing test was originally designed to help identify intelligence - using the principles of the Turing test, researchers aim to find "some evidence of original, intelligent, specific human-like contents" in order to classify activity as human. Applying the Turing test to the identification of Twitter bots, signs of intelligence would indicate that a Twitter account is operated by a human, while a Twitter bot would lack intelligence and thus would not be able to pass a Turing test [5]. The use of this test is difficult in research involving only a few research staff and an extensively large set of data.

In general, it is difficult to obtain a ground truth for more complex bots, for the purpose of research. Without being able to directly verify bot status with the person who created the account, we simply have to use the tools we have available to make a judgement call as to whether an account should be classified as a bot or not.

2 Data Collection

2.1 Challenges in data collection

Twitter is a massive platform with millions of active monthly users generating millions of tweets. In order to obtain a true sense of all bot activity, a massive amount of data would need to be collected. One of the challenges with collecting data is that Twitter places limits on the amount of data that can be collected. Twitter's API allows developers to open a real-time connection that receives up to a 1% subset of the current tweets being sent at a given time. The API also allows one to filter the stream based on keywords of one's choosing that will appear in the tweets returned over the stream. However, the streaming API is also a blackbox - exactly how Twitter determines which tweets to return over this API is unknown. Additionally, standard developer accounts are only allowed to open one stream at a time, thus limiting the overall amount of tweets that can be collected in a given period of time. Our implementation of a Twitter stream saw the return of approximately 350,000 tweets within a twenty-four hour period.

Twitter places much stricter limits on the explicit collection of user data. For example, only the last 200 tweets can be collected for a given user account rather than the entire tweet history. Additionally, this operation is limited to 1,500 requests per fifteen minute interval, as are most of the other API calls Twitter allows. In practice, we were able to collect the timeline, mentions, and user profile of approximately 10,000 accounts every twenty four hours. These implies that collecting all the information related to one million accounts would take more than three months of constant API use.

A more difficult problem to solve is how to collect accurate training data for a model designed to detect political bots. There is no simple or straightforward method to collect verified bot accounts or tweets. Additionally, many datasets that are publically available are dated and may no longer be relevant. Public datasets often lack all of the account information needed, or contain a large percentage of accounts that have been removed by Twitter. The problem of collecting accurate training data is complex, and creates additional problems. Without a dataset of existing accounts determined to be bots, it is impossible to train an algorithm to find additional bots. Larger research studies have made use of manual annotation on thousands of training accounts [8]. With limited resources and study staff, this is not a feasible method for our research. Other research has suggested that there is an average number of daily tweets that makes an account "suspicious". The number of tweets needed to be considered a bot is not settled, but estimates start at 50 tweets per day [16]. However, setting a hard limit on the number of daily tweets for bot versus human classification has several problems.

The use of a predefined daily tweet limit is an enticing proposition because it allows for the quick classification of thousands of bot accounts. The user account information provides the date that each user account was created and its total tweet count. A simple application of division provides the average daily tweet count. This is in fact the first method we employed when attempting to develop a training dataset. The problems with this method were quickly realized after developing our first machine learning model. This results in an algorithm that is hyperfocused on only one aspect of the account. If the daily limit is set high enough, say 144 tweets per day (which equates to one tweet every 10 minutes) as suggested by the Digital Forensic Research Lab [17], it is likely that accounts over this limit are making use of automation, and are thus correctly classified as bots. However, it is certainly possible for automated accounts to be tweeting less than this threshold - as such, using this threshold likely incorrectly classifies a number of automated bot accounts as human. This use of a strict tweet limit fails to provide a broad enough sample of bot accounts to train a general bot detection algorithm. Therefore we were tasked to develop an alternative method in order to gather training data.

2.2 Data collection method

As previously stated, the biggest challenge in data collection is determining how to identify which accounts are bots and which are human for the purpose of our training dataset. Thankfully, we were able to make use of an excellent resource developed by the University of Indiana, called Botometer. Formerly known as BotOrNot, this project was

born out of a competition hosted by the Defense Advanced Research Projects Agency, and can be publically used to classify Twitter accounts as bots or humans.

2.2.1 Botometer

Botometer is a joint project of the Indiana University Network Science Institute and the Center for Complex Networks and Systems Research. Originally launched in 2014 under the name BotOrNot [18], Botometer provides a free-to-use classification system. Botometer requires a Twitter user's entire public profile, last 200 tweets, and all of their recent "@mentions" in order to classify an account as a bot or human. Botometer uses more than 1,000 traits grouped into six categories (content, friend, network, sentiment, temporal, and user meta data) to provide account classification [8]. Botometer's API returns a bot score for each category as well as an overall score, called the Complete Automation Probability (CAP). Botometer defines CAP as:

> "The probability, according to our models, that this account is completely automated, *i.e.*, a bot. This probability calculation uses Bayes' theorem to take into account an estimate of the overall prevalence of bots, so as to balance false positives with false negatives." [19]

CAP Scores range from 0 (definitely human) to 1 (definitely bot). We used Botometer to collect CAP scores for 695,527 Twitter accounts tweeting on political hashtags during May through October of 2018.

2.2.2 Finding Twitter accounts of interest

In developing an algorithm designed to identify political bots, we sought to collect Twitter accounts with a political focus. Our goal was to filter the Twitter stream to

current political hashtags that are likely being utilized by political bots. To determine which hashtags were relevant, we made use of a model developed by two students at the University of California, Berkeley called *BotCheck* - a tool that can be used to identify bots [20][21]. Their tool is specifically focused on political bots and the spread of fake news and misinformation. Their website *BotCheck.me* shows the six most common hashtags currently being used by political bots that their model has detected. We used their list of hashtags to filter the Twitter stream in an effort to find the large number of political bots needed to train our model. Throughout the weeks of data collection, the hashtags that were most popular changed infrequently. Some hashtags such as "#MAGA" and "#QANON" remained common throughout the period of data collection. However, two real-world events triggered a significant change in the most common hashtags used by political bots, according to *BotCheck.* Two events occured that triggered a large bot response during data collection, one of which elicited a much larger bot response and will be a major point of discussion later in this paper.

2.2.3 Technical implementation

In order to amass hundreds of thousands of user profiles, CAP scores, and millions of tweets, data collection needed to occur continuously. To achieve continuous data collection, two Raspberry Pi computers were used.

The first computer collected the Twitter stream (i.e. real-time tweets) and saved details about the user and the tweet in a comma separated value (CSV) file. User information

collected included each account's entire public profile, containing information such as username, screen name, description, tweet count, friend count, account creation date, and more. Tweet information included tweet text, a tweet creation timestamp, retweet count, like count, and additional information. To record the Twitter stream, a Python script using the Tweepy package was created. A stream was started by providing the relevant hashtags to filter for, and Twitter subsequently returned relevant tweets. As previously stated, how Twitter decides which tweets to return over the stream is not known.

We would then go on to use Botometer to classify every user account associated with the tweets collected from the stream as a bot or a human. The second Raspberry Pi was responsible for requesting CAP scores from Botometer. In order to obtain a CAP score for each account from Botometer, the computer first requested each users timeline (last 200 tweets), its complete user profile, and all of its recent mentions (i.e. who is interacting with the account). This information was sent to Botometer, which subsequently returned a CAP score. Due to the nature of Twitter's API rate limits and the amount of information that Botometer requires to provide a CAP score, this operation is severely limited when compared to collecting tweets over the stream. Typical streaming data obtained for a 24 hour period will contain approximately 350,000 tweets. However, obtaining the necessary account information for Botometer to calculate a CAP score is limited by Twitter's API only allowing for retrieval of account information for 12,000 accounts in a twenty-four hour period. This resulted in a 7-10 day period of time required to collect bot scores for all of the users in one day's stream.

In order to improve the efficiency of assigning CAP scores to accounts collected, we designed a system that avoided requesting account information for the same account multiple times. We loaded the streaming CSV, and duplicate user ids were removed such that information requests for each account were performed only once. Additionally, the user ids in the current streaming file were checked against a running list of all accounts previously classified, in order to remove any account for which we already obtained a CAP score. This resulted in one continuously growing CSV file containing a CAP score and user profile information for every account.

3 Creation of Training Datasets

3.1 Botometer determination of human vs. bot

During data collection almost 700,000 unique accounts were observed tweeting and more than 5 million tweets were recorded. Botometer was used to provide a CAP score for all 700,000 accounts in the dataset. Unfortunately, Botometer does not provide a specific cut-off to separate bots and humans, and does not provide guidelines for selecting this value. However, they do caution against setting an arbitrary cut-off. Previous work has established that accounts with a CAP score above 0.53 can be classified as bots [22]. Using this value, only 3.08% of accounts were classified as a bot. The University of Indiana's research estimates that between 9% - 15% of accounts are bots [8], so this is a very conservative bot threshold. Previous research has also established that accounts with

a CAP score below 0.4 can we classified as human [22]. This results in over 95% of accounts being classified as human and leaves a little over 1.5% of accounts unclassified. It might seem logical to set a lower human threshold, however, doing so would expand the gap between human and bot users which would result in more accounts being incorrectly classified. As a result, 0.53 and 0.4 were selected as the thresholds for bot and human, respectively. The distribution of CAP scores is displayed in Figure 2.

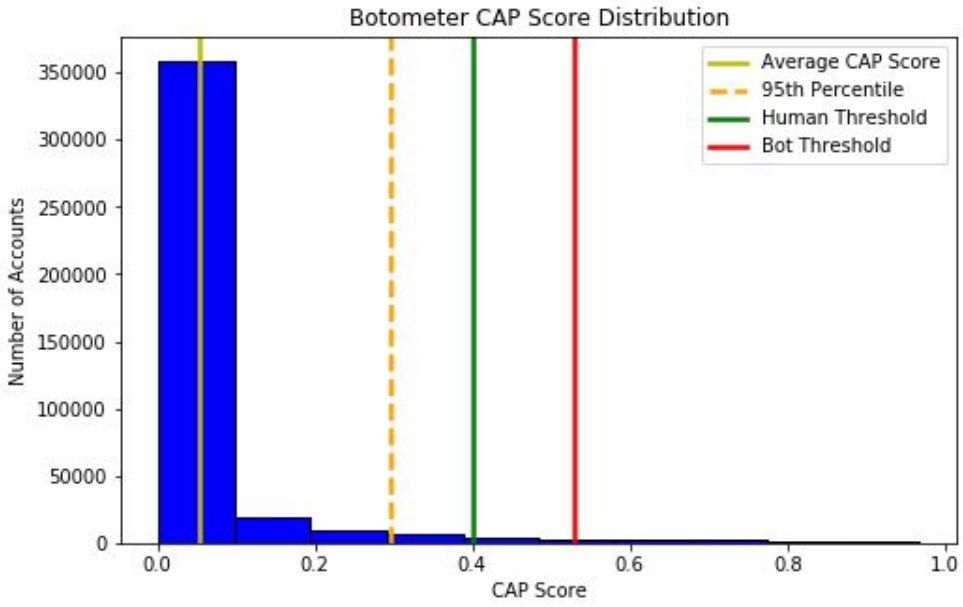


Figure 2: Frequency distribution of the CAP scores obtained from Botometer. Bots are classified as accounts with a minimum cap score of 0.53 (red line). Human accounts have a maximum CAP threshold of 0.4 (green line). Both choices were informed from previous analysis [8].

3.2 Botometer recheck

As previously discussed Botometer provides a CAP score for every account that is requested. However, there is some chance that an account was misclassified by Botometer. Therefore, we decided to recheck the Botometer score of accounts in our dataset in order to determine which accounts were consistently classified as bots.

Checking hundreds of thousands of accounts takes a lot of time when Twitter is imposing API rate limits. However, it is feasible to recheck a smaller portion of accounts. Therefore, we only rechecked the 21,418 accounts that Botometer had already classified as a bot. During the recheck, we discovered that a number of accounts no longer existed. When an account was rechecked, if it no longer existed, Twitter's API would return a 404 error when requesting that user's information. There are two possible reasons for the 404 error - either the user removed their account or Twitter removed their account. Because Botometer previously flagged these accounts as bots, the 404 error likely represents removal by Twitter, essentially validating Botometer's classification.

After the 2016 presidential election Twitter faced significant scrutiny over the use of bots on their platform. As time went on and the scrutiny continued, Twitter finally started to become more vigilant about the problem [13][15]. Thus, if an account was removed prior to our recheck, it is very likely that Twitter detected the bot as well. Moreover, even if the account was removed by the user, this could also be a signal that the account was likely a bot. It is possible that groups operating bots automated the account creation and removal in a timeframe that reduces the likelihood of being detected by Twitter. For the purpose of our research, we argue that removed accounts are more likely to be bots.

Out of our original dataset of 695,527 accounts, only 21,418 accounts were classified as a bot by Botometer. Rechecking these accounts resulted in 9,126 accounts being removed,

which represents 42.6% of previously identified bots. Among this group of removed accounts, CAP scores spanned the entire spectrum of scores in the original set of 21,418 bots, from 0.53 to 0.97. The average CAP score of removed accounts was 0.74, which is slightly higher than the average CAP score of the original set of bots (0.71). This reflects that while Twitter removed bots throughout the spectrum of CAP scores, more bots with higher CAP scores were removed.

In addition to determining the number of removed accounts, we also obtained new CAP score for the 12,291 accounts that were still active on Twitter. This allowed us to determine how consistent CAP scores were in determining which accounts are likely to be bots. Based on our designated bot and human cut-off values of 0.53 and 0.4, respectively, the Botometer recheck demonstrated that out of the 12,291 previously identified bots still active on Twitter, 7,496 accounts remained classified as bots, while 3,378 were reclassified as humans and 1,417 fell between the cut-off values.

Figure 3 shows a scatter plot of old vs. new CAP scores (before and after being rechecked by Botometer) for the 7,496 accounts that remained classified as a bot. The scatter plot demonstrates that CAP scores are most consistent when Botometer has high confidence an account is a bot, while accounts in the lower range tended to have more variability in their CAP scores.

On average, CAP scores for bot accounts were reduced by 0.13. Since Botometer takes so many features into account when generating CAP scores, it is difficult to know exactly what causes reductions in CAP scores. Possible reasons for this include: accounts interacting with more bots during one observation period than another, bot-like tweeting behavior, or changes to their profile data.

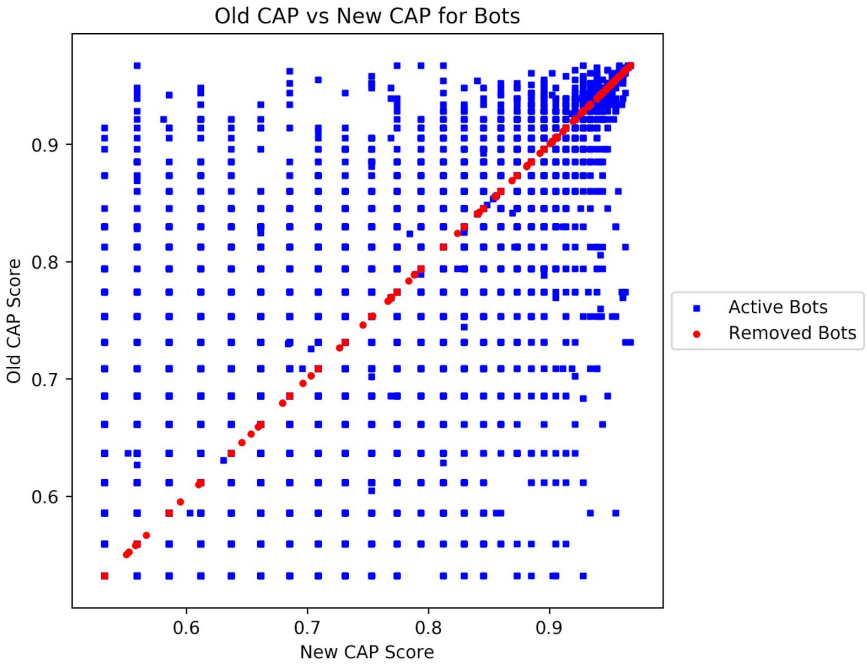


Figure 3: Old vs new CAP scores for bot accounts that were still active on Twitter are shown in blue. Old CAP scores for removed accounts are shown in red, graphed on the line $y = x$. New CAP scores do not exist for removed accounts.

4 Detection Using Machine Learning

4.1 Selection of account features for training

When trying to determine what data to use to classify accounts as bot or human, more data is often regarded as advantageous. Botometer claims to use more than 1,000 features when providing a CAP score [8]. However, users on Twitter do not have access or the time to look at 1,000 features. Aside from the user's name and tweet text, one of the most accessible features of a Twitter account is the user's profile. At the most, this is what users are likely to examine in gauging whether or not an account represents a bot. For this reason, we decided to limit our classification model to only the user's profile information. While this may represent a rudimentary attempt to identify bots, we wanted to explore whether this is a feasible method of bot identification for laypeople browsing Twitter.

4.2 User profile data preprocessing

During data collection, the following information was collected from each account profile: user id, favorites count, statuses count, description, location, creation date, verification status, urls for the account page/profile image/background image, listed count (the number of public lists each user is a member of), followers count, default profile image, friends count, default profile, name, screen name, language, and their geo-enabled status. We were able to quantitatively assess a majority of the aforementioned features as they pertained to the likelihood of being a bot. For a few of these features, the data was collected in such a way that they were not automatically quantifiable - specifically the

user description, the account creation date, and the user's name and screen name. We quantified the user description by calculating the number of characters and the the number of words. We used the creation date to calculate each account's age in days, as of the date on which we observed it tweeting. We quantitatively assessed name and screen name by attempting to assess the level of "gibberish" using natural language processing (specifically via quantifying unique character counts and vowel to consonant ratios). In addition to separately assessing followers count and friends count, we also generated another value for comparison: the followers to following ratio. This is the ratio of the number accounts that a user is following (friends count) to the number of accounts that are following the user (followers count).

However, we did not attempt to quantify user id, location, and urls. Unique values like user id and url for the account page were dropped from the dataset because they are assigned by Twitter and have no relation to bot status. Urls for the profile image and background image were dropped from the dataset due to an inability to adequately quantify these values for comparison between bots and humans. The location was dropped from the dataset because a very high percentage of accounts gathered did not list a location on their profile.

4.3 Results

Several different machine learning algorithms were used when attempting to create the most accurate model: Logistic Regression, Perceptron, Decision Tree, and Random

Forest. Each algorithm was trained on all accounts originally classified as bots (once-classified bots), as well as two subsets of this data after the Botometer recheck:

- Removed bot accounts
- Bot accounts consistently classified as bots by Botometer (twice-classified bots)

We chose to train each algorithm on the subset of removed bot accounts because we argue that removed accounts are more likely to truly be bots, for reasons mentioned earlier in this paper. We also chose to train each algorithm on bot accounts that were classified as bots by Botometer *twice*, both before and after the Botometer recheck. As mentioned earlier, because CAP scores are dynamic and can change over time, we argue that accounts with CAP scores that reflect bot status at two separate time points are more likely to truly be bots. Figure 4 summarizes the results from various algorithms on the three sets of data.

	Accuracy		
Algorithm	Once-classified bot accounts	Removed bot accounts	Twice-classified bot accounts
Logistic Regression	**82.8%**	**88.0%**	**81.8%**
Decision Tree	**89.0%**	**96.1%**	**95.8%**
Random Forest	**89.9%**	**97.4%**	**96.3%**
Perceptron	**75.1%**	**88.3%**	**68.8%**

Figure 4: Accuracy of various machine learning algorithms, using Botometer and a minimum bot CAP threshold of 0.53 as the reference standard.

Figure 4 demonstrates that the Random Forest model was consistently the most accurate in predicting which accounts were bots, over all three sets of data. It also demonstrates that training each algorithm on the subset of removed bot accounts appears to result in the

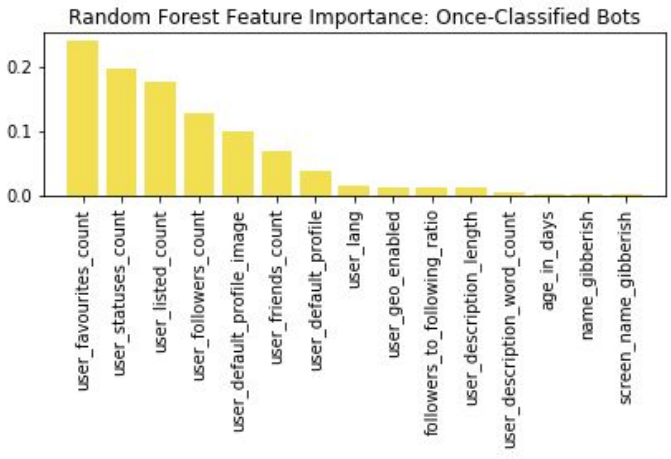
highest classification accuracy, with a 5.2 - 13.2% improvement over using once-classified bot accounts. This is not surprising, as this set of data represents accounts that were determined by Botometer to be bots, and were subsequently removed by Twitter, presumably for this reason.

Because the Random Forest demonstrated the highest accuracy for all three sets of data, our remaining discussion will focus on this algorithm. Figure 5 demonstrates a breakdown of the importance of each feature when using the Random Forest model for each of the three sets of data. The figure demonstrates that the number of tweets a user favorited (liked) consistently has the largest predictive value in classifying an account as a bot or human using a Random Forest. Human users had an average of over 15,000 favorites while bots had an average of only 3,100 favorites. This drastic difference allows for this value to have such a large significance when classifying accounts as bot or human. This was surprising because just like re-tweeting, favoriting a tweet is easily automated.

The number of tweets was unsurprisingly influential when classifying accounts as bot or human. It is known that bots on average are more active than human users - a large number of posts per day may represent a high level of automation [3]. Our data supports this notion, as bots tweeted on average 27.4 times per day, while humans tweeted on average 20.0 times per day. We suspect that the greater number of tweets from bots represents a greater number of re-tweets of other users rather than original tweets, as

re-tweeting other users is easier to automate. Overall, we were very pleased with our results, considering the limited amount of information our model uses to classify accounts as bots versus humans.

Other features that appeared to be relatively predictive of whether an account is a bot or human include: listed count (12.3%), followers count (10.0%), default profile image (6.3%), friend count (6.2%), default profile (3.8%), and user language (3.4%). On the other hand, features that appeared to be less significant included geo-enabled status (1.7%), followers to following ratio (0.7%), description length (0.5%), description word count (0.4%), account age (<0.1%), and gibberish in name and screen name (<0.1).



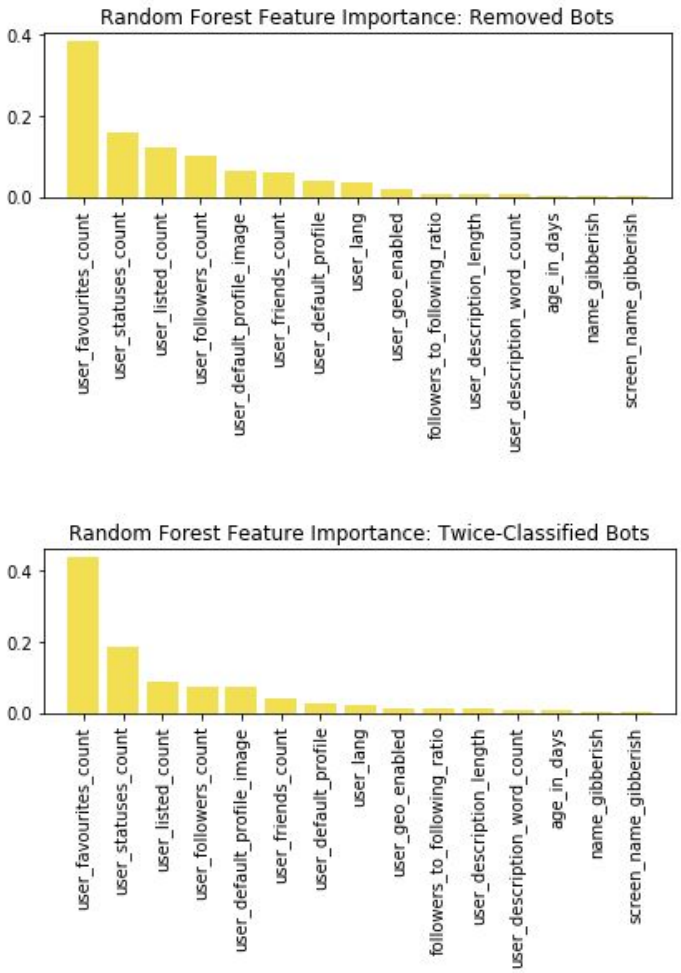


Figure 5: Importance of each user profile feature when using Random Forest to predict whether a Twitter account is a bot, across three datasets: once-classified bots, removed bots, and twice-classified bots.

4.4 Impact of bot and human thresholds

Based on previous research, we set a minimum CAP score for bot classification at 0.53 and were able to achieve 89.9% accuracy with our model using only information contained in an account's Twitter profile. Creating a higher CAP score resulted in increased accuracy of our model. For example using a minimum CAP score of 0.70

resulted in 93% accuracy on our training data. This result is to be expected, due to the fact that as CAP scores increase, the likelihood of being a bot account rises. The profiles of accounts that have higher CAP scores are more obvious bots. Our observations during data analysis demonstrate that bots often only retweet, generate lots of spam, have random looking names, and no profile picture (see Appendix 3 for an example). These accounts are more easily identified because of these traits. It therefore seems logical that a model which only examines profile data would more accurately identify accounts that put less effort into disguising the fact they are a bot.

Interestingly, lowering the human threshold did not have as significant of an impact on classification accuracy. Lowering the human threshold from a max CAP of 0.4 to a max CAP of 0.30 only resulted in a 0.3% increase in the accuracy of our model. This suggests that the differences between accounts in the 0.3-0.4 range are not as significant as the differences between accounts in the 0.53-0.70 range.

There is a cost associated with raising the minimum CAP score for bot accounts and lowering the maximum CAP score for human accounts. Doing so creates a larger group of accounts that the model hasn't seen and thus would be difficult for it to classify accurately. While adjusting the CAP thresholds in this way may increase the accuracy of the model on the training data, this increased accuracy is unlikely to carry over into real-world classifications, as many accounts will fall in the now expanded "unknown"

region between the human and bot CAP thresholds. For this reason, we do not advise setting a lower human or higher bot threshold.

4.5 Limitations

The limitations of this research stems from the fact that the collection of training data is very difficult. Even if accounts were manually collected and annotated, the accuracy of those classifications would be limited by how well researchers were able to classify said accounts. Botometer made use of multiple researchers to classify each account in their collection of training data, thus reducing the likelihood that any single account is misclassified [8]. In our study, we relied on Botometer for the classification of training data. As such, if Botometer misclassifies an account, this error translates into our model and affects its accuracy.

There are additional limitations on the application of our model. Accounts with CAP scores falling between the 0.40 and 0.53 fall into a range that makes it nearly impossible to determine bot status. This is because, in addition to these accounts receiving middle-of-the-road scores, the *truth* of whether an account is a bot is impossible to know for sure - knowing this would allow for more precise training. Even with manual classification, it may remain extremely difficult to determine bot status, especially for accounts with middle-of-the-road, non-extreme CAP scores. Therefore, it is difficult to say how accurate our algorithm would be in the real world, since it is most likely to misclassify accounts in this range.

However, there are other limitations that extend beyond just the classification model. The University of Indiana estimates that anywhere between 9 percent to 15 percent of all Twitter accounts are bots [8]. However, other research places this number as high as 35 percent [4]. During this study, only around three percent of all accounts observed were bots. There are several reasons why this result might not be an accurate estimate of the true percentage of bots on Twitter. First and foremost, only certain political hashtags were observed and the hashtags we chose to observe were based on previous research [20][21]. Therefore, the results outlined here are only applicable to accounts tweeting on political hashtags. It is likely that a more generic collection of tweets could result in a higher percentage of accounts being classified as bots. After all, not all bots are political bots. Moreover, it is possible that the number of bots operating at a given period of time is dependent on real-world events. It is extremely likely that during a US presidential election, more than three percent of accounts using political hashtags would be classified as a bot. Additionally, it is also possible that the reduced percentage of bots is a result of Twitters increased action to combat bots on their platform [13][24][25]. This research found that 42% percent of bot accounts were removed during the data collection period, so it is likely that the overall percentage of bots on Twitter's platform is declining. However, it is difficult to truly determine if this is the case, due to limits on feasible data collection and the size of Twitter's user base, which is estimated to be more than 67 million active monthly accounts in the United States alone [30].

5 Bot Response to Political Events

5.1 Hashtag shifts with political events

US politics has not been without turmoil for much of 2018. Sample data collection began in May of 2018, a time without major scheduled political events (i.e. an election). During the course of data collection, however, we noticed that minor events began to attract the focus of political bots on Twitter. As an example, ABC's firing of Roseanne Barr and its discontinuation of the reboot television series *Roseanne* generated a large political bot response, evidenced by trending bot hashtags on the website *BotCheck.me*. Due to the nature of Roseanne Barr's political views and President Trump's vocal response against the firing, the event became political in nature, explaining the response by political bots. While this response was rather short-lived, the event and its subsequent Twitter response demonstrated that political bots can and will respond quickly to real-world events. As such, we continued to monitor the most popular bot hashtags on *Botcheck.me*, and how they shifted in response to real-world political events.

In the middle of September 2018, sexual assault allegations against Supreme Court nominee Brett Kavanaugh were made public in *The New Yorker* magazine. Shortly after this event, *BotCheck* demonstrated that political bots began to tweet about Brett Kavanaugh's nomination. We collected tweets during the height of the controversy, from September 23rd, 2018 until two days after the Senate's confirmation vote October 8th, 2018. Numerous hashtags became popular during this time (see Appendix 2). Some of these hashtags were inherently supportive of Kavanaugh's nomination, such as

"#ConfirmKavanaugh" or "#ConfirmKavanaughNow". Others such as "#Kavanaugh", "#KavanaughHearings", and "#JusticeKavanaugh" remained more neutral. Other hashtags that became trending, such as "#Winning" appeared to be unrelated, but in the context of the confirmation, could be argued to represent support of Kavanaugh. Given the black-box nature of Twitter's streaming API, it is difficult to determine the intent of tweets using such hashtags. Regardless, during this time period, data collection was performed with the intention of capturing accounts with tweets containing hashtags related to the Brett Kavanaugh Supreme Court nomination, as shown in Appendix 2.

5.2 Bot activity during Brett Kavanaugh Supreme Court nomination

Out of almost 700,000 accounts for which CAP scores were collected, 287,307 accounts were collected during the two weeks of controversy around Brett Kavanaugh's nomination to the United States Supreme Court, from September 23rd to October 8th, 2018. During this time, these accounts were observed sending more than 412,000 tweets. Bots accounted for 4.02% of all tweets observed during this time period. This represents an increase of almost 1% from bot tweeting activity observed prior to the Kavanaugh controversy, at 3.17%. Additionally, 3.98% of all accounts observed tweeting during this time were classified as a bot by Botometer. This represents an absolute increase in bot accounts of 1.53%, as bots represented only 2.45% of accounts prior to this time period of political controversy. The average account age of bots active during this time was 613 days, compared to 698 days prior to the Kavanaugh controversy. Because the average account age decreased, we can conclude that during this time period, new and young

accounts emerged that began tweeting about this specific political issue. However, whether these young accounts emerged with the sole intention of focusing on this political event cannot be determined. Because the average account age during this time was 613 days (still a relatively high age), we can conclude that bot accounts which had already existed prior to these two weeks began to shift their focus onto this issue.

6 Twitter Removal

6.1 What is Twitter doing about political bots?

As previously discussed, Twitter has become more active in its efforts to combat bot activity on its platform [13]. After our data collection, we were unable to assign CAP scores to approximately 24,000 accounts because between the time of account collection and the time CAP scores were assigned, we noticed that these accounts no longer existed (resulting in only 695,527 accounts in our training dataset). For the purpose of our research, we presume that the majority of these nonexistent accounts were removed by Twitter, with the most plausible rationale being that they are bots. While this is a significant number of presumably removed Twitter bots, we still discovered over 21,000 bots within our remaining accounts. Thus, there still exist some very real concerns about Twitter's actions to combat political bot activity.

In evaluating Twitter's response to political bot activity, we became interested in determining how quickly Twitter is able to identify and remove bots. We sought to

determine this by examining the average account age of accounts (presumably bots) removed from Twitter. After our initial data collection, we identified 21,418 accounts as bots using Botometer. As described earlier in this paper, we then decided to re-check Botometer scores for these accounts some time later and discovered that 9,126, or 42% of our once-classified bots were presumed to be removed by Twitter due to suspicion of bot activity. These removed bots had an average CAP score of 0.73, an average account age of 228 days, and sent an average of 3,959 tweets over their lifespan. While this may suggest that it takes over 200 days for Twitter to identify and remove political bots, there are legitimate issues with criticizing Twitter's response time using account age as a proxy. We need to consider that it is possible for existing bot accounts to have previously represented authentic accounts, due to the fact that accounts can be hacked and used as bots. These previously authentic bot accounts may be more difficult for Twitter to identify, due to existing humanoid profile information and activity on these accounts for a lengthy period of time. Account age is also not a reliable proxy for Twitter's response time due to the fact that bot accounts may remain dormant for extended periods of time. With little to no activity during these dormant periods (during which the account age is driven up), it may be difficult for Twitter to identify these bots. Because of the aforementioned issues, we cannot use account age as a proxy for the time it takes for Twitter to identify and remove bots.

We think that while Twitter has increased their efforts to combat political bots on their platform, it can be argued that not enough is being done. While 9,126 once-classified bots

were removed by Twitter during the course of data analysis, after re-checking Botometer scores, 7,496 bots remained active on Twitter. Moreover, if we were able to detect thousands of bots while only using two low-priced computers with limited data collection, it seems feasible for a large company like Twitter to be able to detect and remove more bots than it is currently detecting and removing. Moreover, our research supports the notion that thousands of new accounts are being created in response to a political events, despite the fact that recently, Twitter changed its policy to require that all new applications go through an approval process [7]. Additionally, Twitter's value is based largely on the number of active accounts on their platform. Therefore, it would seem that Twitter has a conflict of interest when it comes to the large scale removal of political bot accounts. As such, it is possible that Twitter may be slow-rolling this purge of political bot accounts in order to soften the financial blow. Nevertheless, Twitter is more active than ever when it comes to the removal of bots on their platform. In fact, in the last financial quarter alone, the company has purged millions of fake accounts [29].

6.2 Bot-detection bot

During data collection, we decided to create our own Twitter bot. Instead of creating a political Twitter bot, we created an informative bot with the purpose of sharing the identity of political bots we identified in our research. The code required to send out automated tweets was surprisingly simple (see Appendix 1). Whenever Botometer returned a CAP score over 0.70 for a given Twitter account, our automated account named "BotDetectionBot" would send out a tweet, as shown in Figure 1.

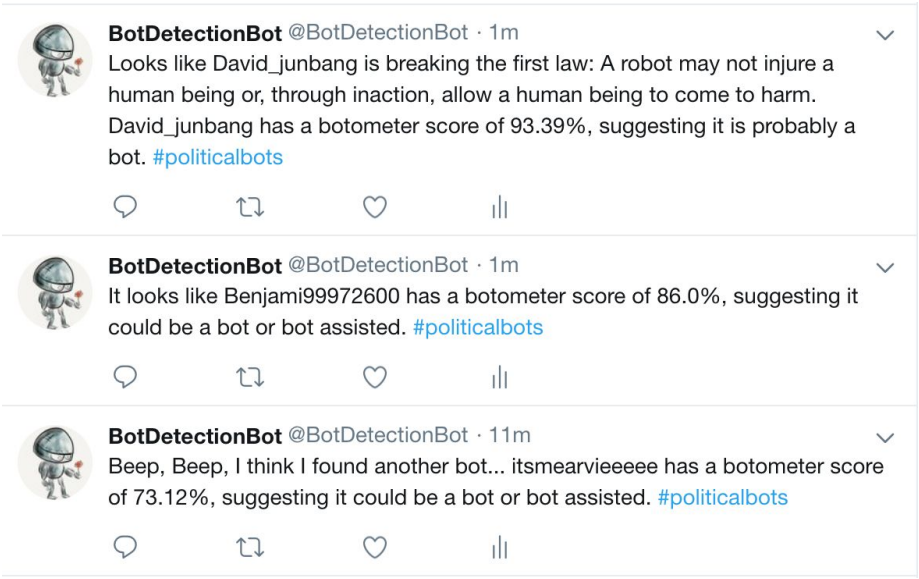


Figure 1: Sample tweets sent from created Twitter bot on October 17th, 2018.

Our bot account was suspended several times by Twitter. The first account suspension was the most severe (the entire account was suspended) and the quickest. Originally, our Twitter bot would @ mention the bot accounts in the tweet text - automated @ mentioning another account without previous interaction with it is against Twitter's Automation rules [23]. Within the first 40 tweets, our account was banned. As such, we created a completely new bot on Twitters platform. Following this event, our bot's ability to tweet was revoked five times by Twitter - the bot would attempt to post tweets, but Twitter would not display them on the platform. Only after manually logging in and verifying our account would Twitter restore our bot's ability to post tweets. This account verification required that a code sent via SMS be entered on Twitter's website. Twitter's response to my bot was surprisingly quick, especially when compared to the lifespan of the bots we observed in our research. We suspect that the use of repetitive, generic tweet content, and the highly automated nature of BotDetectionBot made it easier for Twitter to

detect. To date, BotDetectionBot has sent out over 13,000 tweets identifying political bots.

7 Conclusion

7.1 Future work

We advocate for two areas of future research related to our work. First, we argue for additional research into improving our model's accuracy. Second, is a complete shift in the focus of research currently related to the identification of political bots on Twitter.

In order to increase the accuracy of our model, other features should be extracted that currently are not captured within our model, such as the user's profile image and background image. Currently our model is only able to determine whether or not the user has a default profile image (meaning no image has been uploaded). However, many bots are created using stock images available on the Internet. Using a reverse Google image search can allow for easy confirmation that an image is not original. Incorporating this feature into our model would allow us to determine when an account has a "fake" user image, which can provide helpful information in distinguishing between humans and bots. It is unclear how valuable this information would be - regardless, this is an unexplored feature that may prove to be helpful.

Our research focuses on identifying automated political bot accounts. However, the truth of the matter is that not all “bots” are fully automated - the term “bot” is commonly used to represent “fake” accounts [4]. Non-automated political bots include accounts created and operated by humans, but share the same goal as automated political bots - to promote specific political interests by spreading misinformation. Therefore, it seems that the best way to identify political bots (or “fake” accounts) is to shift the focus from the appearance and behavior of an account to the content shared by the account. While identifying automated political bots is challenging (because ground truth is difficult to obtain), factually inaccurate information is very objective and can be easily identified by machine learning. Moreover, while automated political bots can become more difficult to identify as technology becomes more sophisticated, bot-detection models can be updated more easily to account for new forms of misinformation [26][27]. Additionally, research has shown that fake news is actually spreading faster than real news on Twitter [28].

Moreover, the reason for increasing interest in identifying political bots centers around the spread of misinformation. Thus, a shift to focus on the actual content being shared by Twitter accounts would allow for more direct detection of what we consider as political bots.

7.2 Conclusions

In this paper, we have outlined a method for collecting training data, quantified the presence of political bots on Twitter’s platform, demonstrated that machine learning can be used to identify political bots with limited data, and evaluated Twitter’s response to

this growing problem. Based on our collection method, we were able to estimate that political bots make up about 3-8% of accounts tweeting on political hashtags between May and October of 2018. Using these data, we were able to design a model that achieves approximately 97.4% accuracy when classifying bots. Additionally, we were able to demonstrate that Twitter is certainly taking steps to address this problem. It removed 9,126 accounts that we originally identified as bots. Our research is limited by the difficulty of collecting training data, and the fact that new data needs to constantly be collected in order for our model to remain up-to-date with the dynamic nature of bots. It is unlikely that our model will be as accurate as it currently is in a year's time. Therefore, we recommend that future work focus on the identification of misinformation, as doing so is easier due to the objectivity of inaccurate information. This would allow for the identification of not only automated bot accounts, but also non-automated, fake accounts operated by real people, enabling Twitter to combat the spread of misinformation on its platform.

8 References

[1] Dunham, K., & Melnick, J. (2009). *Malicious bots: An inside look into the cyber-criminal underground of the internet*. Boca Raton, Fla: CRC Press.

[2] Zeifman, I. (2017, January 24). Bot Traffic Report 2016. Retrieved November 25, 2018, from https://www.incapsula.com/blog/bot-traffic-report-2016.html

[3] Kollanyi, B., Howard, P., & Woolley, S. (2016). Bots and Automation over Twitter during the First U.S. Presidential Debate. Retrieved September 10, 2018, from http://comprop.oii.ox.ac.uk/wp-content/uploads/sites/89/2016/11/Data-Memo-US-Election.pdf

[4] Velayutham, T., & Tiwari, P. K. (2017). Bot identification: Helping analysts for right data in twitter. *2017 3rd International Conference on Advances in Computing,Communication & Automation (ICACCA) (Fall)*. Retrieved September 11, 2018.

[5] Chu, Z., Gianvecchio, S., Wang, H., & Jajodia, S. (2012). Detecting Automation of Twitter Accounts: Are You a Human, Bot, or Cyborg? *IEEE Transactions on Dependable and Secure Computing,9*(6), 811-824.

[6] Petre, C. (2017). Pax Technica: How the Internet of Things May Set Us Free or Lock Us UpPax Technica: How the Internet of Things May Set Us Free or Lock Us Up, by HowardPhilip N.New Haven, CT: Yale University Press, 2015. 320 pp. $28.00 cloth. ISBN: 9780300199475. *Contemporary Sociology: A Journal of Reviews,46*(1), 84-85.

[7] Twitter apps - Twitter Developers. (n.d.). Retrieved from https://developer.twitter.com/en/docs/basics/developer-portal/guides/apps.html

[8] Varol, O., Ferrara, E., David, C., Menczer, F., & Flammini, A. (2017). Online Human-Bot Interactions: Detection, Estimation, and Characterization. Retrieved September 10, 2018, from https://arxiv.org/abs/1703.03107.

[9] Howard, P. N., & Kollanyi, B. (2016). Bots, #Strongerin, and #Brexit: Computational Propaganda During the UK-EU Referendum. *SSRN Electronic Journal*.

[10] Howard, S. W. (2017, June 03). Bots Unite to Automate the Presidential Election. Retrieved from https://www.wired.com/2016/05/twitterbots-2/

[11] Chen, A. (2015, June 02). The Agency. Retrieved from https://www.nytimes.com/2015/06/07/magazine/the-agency.html

[12] Forelle, M. C., Howard, P. N., Monroy-Hernandez, A., & Savage, S. (2015). Political Bots and the Manipulation of Public Opinion in Venezuela. *SSRN Electronic Journal*.

[13] Russell, J. (2018, February 22). Twitter is (finally) cracking down on bots. Retrieved from https://techcrunch.com/2018/02/22/twitter-is-finally-cracking-down-on-bots/

[14] Automation and the use of multiple accounts. (n.d.). Retrieved from https://blog.twitter.com/developer/en_us/topics/tips/2018/automation-and-the-use-of-multiple-accounts.html

[15] Update on Twitter's review of the 2016 US election. (n.d.). Retrieved from https://blog.twitter.com/official/en_us/topics/company/2018/2016-election-update.html

[16] Gallacher, J., Kaminska, M., Kollanyi, B., & Howard, P. (2017). Junk News and Bots during the 2017 UK General Election: What Are UK Voters Sharing Over Twitter? Retrieved September 23, 2018.

[17] #BotSpot: Twelve Ways to Spot a Bot – DFRLab – Medium. (2017, August 28). Retrieved from https://medium.com/dfrlab/botspot-twelve-ways-to-spot-a-bot-aedc7d9c110c

[18] David, C., Ferrara, E., Flammini, A., Varol, O., & Menczer, F. (2016). BotOrNot: A System to Evaluate Social Bots. Retrieved September 23, 2018.

[19] Botometer by OSoMe. (n.d.). Retrieved September 24, 2018, from https://botometer.iuni.iu.edu/#!/faq#what-is-cap

[20] Smiley, L. (2017, November 01). The College Kids Doing What Twitter Won't | Backchannel. Retrieved May 1, 2018, from https://www.wired.com/story/the-college-kids-doing-what-twitter-wont/

[21] Labs, R., & RoBhat Labs. (2017, October 31). Identifying Propaganda Bots on Twitter – RoBhat Labs – Medium. Retrieved May 1, 2018, from https://medium.com/@robhat/identifying-propaganda-bots-on-twitter-5240e7cb81a9

[22] Pozzana, I., & Ferrara, E. (n.d.). Measuring bot and human behavioral dynamics. Retrieved June 1, 2018.

[23] Automation rules. (n.d.). Retrieved June 7, 2018, from https://help.twitter.com/en/rules-and-policies/twitter-automation

[24] Griffin, A. (2018, July 13). Donald Trump loses hundreds of thousands of followers after latest Twitter change. Retrieved October 17, 2018, from https://www.independent.co.uk/life-style/gadgets-and-tech/news/donald-trump-twitter-account-followers-fake-account-purge-a8445461.html

[25] Vavra, S. (1969, December 31). Report: Twitter suspended 1,500 for spreading wrong election date. Retrieved October 17, 2018, from https://www.axios.com/report-twitter-suspended-1500-fake-accounts-spreading-false-election-day-information-liberals-30d60651-4c20-4aac-a11a-d0975a013b71.html

[26] Ajao, O., Bhowmik, D., & Zargari, S. (2018). Fake News Identification on Twitter with Hybrid CNN and RNN Models. *Proceedings of the 9th International Conference on Social Media and Society - SMSociety 18*. Retrieved October 30, 2018.

[27] Liu, Y., & Brook Wu, Y. (2018). Early Detection of Fake News on Social Media Through Propagation Path Classification with Recurrent and Convolutional Networks. *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*. Retrieved October 30, 2018.

[28] Dizikes, P., & MIT News Office. (2018, March 08). Study: On Twitter, false news travels faster than true stories. Retrieved from http://news.mit.edu/2018/study-twitter-false-news-travels-faster-true-stories-0308

[29] Cherney, M. A. (2018, October 28). Twitter earnings: More revenue from declining user base. Retrieved October 30, 2018, from https://www.marketwatch.com/story/twitter-pries-more-revenue-from-declining-user-base-2018-10-25

[30] Twitter MAU in the United States 2018 | Statistic. (n.d.). Retrieved from https://www.statista.com/statistics/274564/monthly-active-twitter-users-in-the-united-states/

Appendix

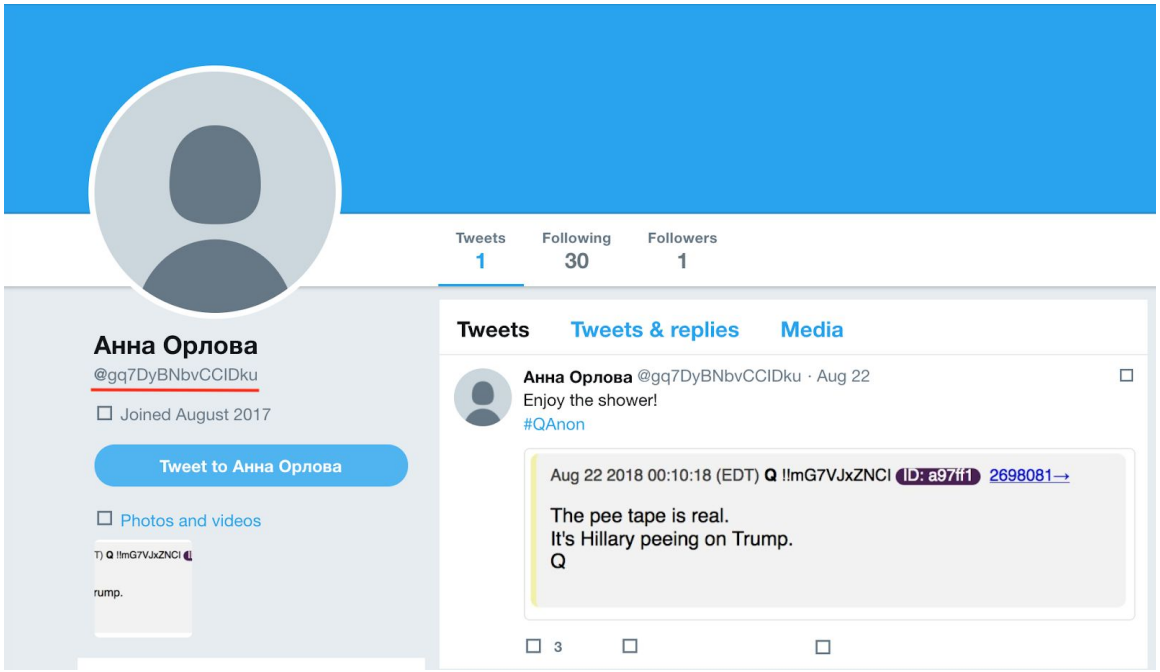
1 Twitter bot tweeting code

```
def send_tweet(self, user, cap):
   text_options = ['Beep, Beep, I think I found another bot...{0}'.format(user),
                   'R2 says {0}'.format(user),
                   'It looks like {0}'.format(user),
                   'I\'ve calculated that {}'.format(user)
                   ]

   cap *= 100
   cap = round(cap, 2)

   start = random.choice(text_options)
   ending = ' has a botometer score of {0}%, suggesting it is probably a
bot'.format(cap)

   tweet_text = "{0}{1}".format(start, ending)
   self.tweepy_api.update_status(tweet_text)
   return
```

2 Hashtags related to Brett Kavanaugh nomination

- #Kavanaugh
- #ConfirmKavanaugh
- #ConfirmKavanaughNow
- #KavanaughHearings
- #JusticeKavanaugh
- #KavanaughConfirmed
- #Winning
- #WalkAway

3 Obvious bot example profile



4 Kavanaugh bot example

5 Record Twitter stream code

```
import constants # API Keys
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream
import time
import csv
import sys
import smtplib


# Create a streamer object
class TwitterStreamListener(StreamListener):

   # Define a function that is initialized when the miner is called
   def __init__(self, hashtags, time_limit=None, api=None):
       super(TwitterStreamListener, self).__init__()
       # That sets the api
       self.api = api

       if time_limit is not None:
           self.has_time_limit = True
           self.start_time = time.time()
           self.time_limit = time_limit
       else:
           self.has_time_limit = False

       hashtag_string = ''

       if hashtags is not None:
           for hashtag in hashtags:
               if hashtag != hashtags[-1]:
                   hashtag_string += '-'
                   hashtag_string += hashtag
               else:
```

```
            hashtag_string += '-'
            hashtag_string += hashtag
            hashtag_string += '-'

# Create a file with 'StreamData' hashtags and the current time
self.stream_filename = 'StreamData' + hashtag_string +
                                      time.strftime('%Y%m%d-%H%M%S') + '.csv'

# Create a new file with that filename
stream_file = open(self.stream_filename, 'w')

# Create a csv writer
stream_writer = csv.writer(stream_file)
```

```
# Write a single row with the headers of the columns
stream_writer.writerow(['status_text',
                        'status_created_at',
                        'status_lang',
                        'status_place',
                        'status_coordinates',
                        'status_id',
                        'status_favorite_count',
                        'status_retweeted',
                        'status_source',
                        'status_favorited',
                        'status_retweet_count',
                        'status_hashtags',
                        'status_hashtag_count',
                        'status_entities',
                        'user_favourites_count',
                        'user_statuses_count',
                        'user_description',
                        'user_location',
                        'user_id',
                        'user_created_at',
                        'user_verified',
                        'user_following',
                        'user_url',
                        'user_listed_count',
                        'user_followers_count',
                        'user_default_profile_image',
                        'user_utc_offset',
                        'user_friends_count',
                        'user_default_profile',
                        'user_name',
                        'user_lang',
                        'user_screen_name',
                        'user_geo_enabled',
                        'user_profile_background_color',
                        'user_profile_image_url',
                        'user_time_zone'
```

```
                            ])

    # When a tweet appears
    def on_status(self, status):

        # Check the time limit
        if self.has_time_limit and (time.time() - self.start_time) > self.time_limit:
            print('Shutting down stream after ', self.time_limit, ' seconds!')
            TwitterStreamListener.send_notification_email()
            # Returning False closes the stream
            return False

        # Open the csv file created previously
        stream_file = open(self.stream_filename, 'a')

        # Create a csv writer
        stream_writer = csv.writer(stream_file)

        hashtags = TwitterStreamListener.parse_hashtags(status.entities['hashtags'])

        try:
            # Write the tweet's information to the csv file
            stream_writer.writerow([status.text,
                                    status.created_at,
                                    status.lang,
                                    status.place,
                                    status.coordinates,
                                    status.id,
                                    status.favorite_count,
                                    status.retweeted,
                                    status.source,
                                    status.favorited,
                                    status.retweet_count,
                                    hashtags,
                                    len(status.entities['hashtags']),
                                    status.entities,
                                    status.user.favourites_count,
                                    status.user.statuses_count,
                                    status.user.description,
                                    status.user.location,
                                    status.user.id,
                                    status.user.created_at,
                                    status.user.verified,
                                    status.user.following,
                                    status.user.url,
                                    status.user.listed_count,
                                    status.user.followers_count,
                                    status.user.default_profile_image,
                                    status.user.utc_offset,
                                    status.user.friends_count,
                                    status.user.default_profile,
                                    status.user.name,
                                    status.user.lang,
                                    status.user.screen_name,
                                    status.user.geo_enabled,
                                    status.user.profile_background_color,
                                    status.user.profile_image_url,
                                    status.user.time_zone
                                    ])

            # print('Added streaming tweet: ', status.text, ' by: ', status.user.name)

        # If some error occurs
        except Exception as e:
            # Print the error
            print(e)
```

```
        # and continue
        pass

    # Close the csv file
    stream_file.close()

    return

# When an error occurs
def on_error(self, status_code):
    # Print the error code
    print('Encountered error with status code:', status_code)

    # If the error code is 401, which is the error for bad credentials
    if status_code == 401:
        # End the stream
        return False

# When a deleted tweet appears
def on_delete(self, status_id, user_id):

    # Print message
    print("Delete notice")

    # Return nothing
    return



# When reach the rate limit
def on_limit(self, track):

    # Print rate limiting error
    print("Rate limited, continuing")

    # Continue mining tweets
    return True

# When timed out
def on_timeout(self):

    # Print timeout message
    print(sys.stderr, 'Timeout...')

    # Wait 10 seconds
    time.sleep(10)

    # Return nothing
    return

####################
# Static Functions #
####################

@staticmethod
def parse_hashtags(hashtag_dict):
    if len(hashtag_dict) > 0:
        hashtag_text = ''
        for dictionary in hashtag_dict:
            if 'text' in dictionary:
                if hashtag_text != '':
                    hashtag_text += ' ' + dictionary['text']
                else:
                    hashtag_text += dictionary['text']
    else:
        hashtag_text = ''
```

```
        return hashtag_text

    # Create a mining function
    @staticmethod
    def start_mining(hashtags=None, time_limit=None):
        """
        :param hashtags: list of strings
        :param time_limit: number of seconds until stream shuts down
        :return: Returns tweets containing those strings.
        """

        # Create a listener
        listener = TwitterStreamListener(hashtags, time_limit)

        # Create authorization info
        auth = OAuthHandler(constants.consumer_key, constants.consumer_secret)
        auth.set_access_token(constants.access_token, constants.access_token_secret)

        # Create a stream object with listener and authorization
        stream = Stream(auth, listener)

        # Run the stream object using the user defined queries
        if hashtags is not None:
            stream.filter(track=hashtags, stall_warnings=True, async=True)
        else:
            stream.sample()


    ######################
    # Email Notification #
    ######################

    @staticmethod
    def send_notification_email():
        # Email myself when the script finishes so I can start on the next set of data
        server = smtplib.SMTP('smtp.gmail.com', 587)
        server.starttls()
        server.login(constants.email_address, constants.password)

        subject = 'Twitter Stream'
        text = 'Streaming Script Finished!'
        message = 'Subject: {}\n\n{}'.format(subject, text)
        server.sendmail(constants.email_address, constants.real_email, message)
        server.quit()

        return
```

6 Start Twitter stream code

```
import streaming # TwitterStreamListener
import sys

# Sample call python3 start_stream.py '#maga' '#qanon' '#roseanne'
# Sample call python3 start_stream.py

# Arguments passed in
hashtags = sys.argv[1:]
if hashtags:
    # Filter the stream on the provided hashtags
    if hashtags[0] == 'help':
        print('You can start streaming tweets on hashtags by making the following call with
                your desired hashtags:')
        print('Sample call python3 start_stream.py \'#maga\' \'#qanon\' \'#roseanne\'')
        print('\n')
```

```
        print("You can start streaming tweets without a filter using: ")
        print('Sample call: \'python3 start_stream.py\'')
        print('\n')
        print("You can also pass in an optional time limit in seconds as the last value")
        print('Sample call: \'python3 start_stream.py 86400\'')
        print('\n')
    else:
        try:
            time_limit = int(hashtags[-1])
            del hashtags[-1]
            # remove time limit from hashtags so it doesn't get put in the file name
            streaming.TwitterStreamListener.start_mining(hashtags, time_limit=time_limit)
            print('Starting stream with ', time_limit, ' second limit!')
        except ValueError:
            # Not an int so no time limit was passed in
            streaming.TwitterStreamListener.start_mining(hashtags)
            print('Starting stream without a time limit!')
else:
    # Take a sample with no filtering
    streaming.TwitterStreamListener.start_mining()
```

7 Code to check accounts using Botometer

```
from __future__ import print_function
import time

import requests
from requests import ConnectionError, HTTPError, Timeout
import tweepy
from tweepy.error import RateLimitError, TweepError


class NoTimelineError(ValueError):
    def __init__(self, sn, *args, **kwargs):
        msg = "user '%s' has no tweets in timeline" % sn
        super(NoTimelineError, self).__init__(msg, *args, **kwargs)


class Botometer(object):
    _TWITTER_RL_MSG = 'Rate limit exceeded for Twitter API method'

    def __init__(self,
                 consumer_key, consumer_secret,
                 access_token=None, access_token_secret=None,
                 mashape_key=None,
                 **kwargs):
        self.consumer_key = consumer_key
        self.consumer_secret = consumer_secret
        self.access_token_key = self.access_token = access_token
        self.access_token_secret = access_token_secret
        self.wait_on_ratelimit = kwargs.get('wait_on_ratelimit', True)

        self.mashape_key = mashape_key

        if self.access_token_key is None or self.access_token_secret is None:
            auth = tweepy.AppAuthHandler(
                self.consumer_key, self.consumer_secret)
```

```
    else:
        auth = tweepy.OAuthHandler(
            self.consumer_key, self.consumer_secret)
        auth.set_access_token(
            self.access_token_key, self.access_token_secret)

    self.twitter_api = tweepy.API(
        auth,
        parser=tweepy.parsers.JSONParser(),
        wait_on_rate_limit=self.wait_on_ratelimit,
        retry_count=3,
        retry_delay=3,
        wait_on_rate_limit_notify=True
        )

    self.api_url = kwargs.get('botometer_api_url',
                              'https://osome-botometer.p.mashape.com')
    self.api_version = kwargs.get('botometer_api_version', 2)

@classmethod
def create_from(cls, instance, **kwargs):
    my_kwargs = vars(instance)
    my_kwargs.update(kwargs)
    return cls(**my_kwargs)




def _add_mashape_header(self, kwargs):
    if self.mashape_key:
        kwargs.setdefault('headers', {}).update({
            'X-Mashape-Key': self.mashape_key
        })
    return kwargs

def _bom_get(self, *args, **kwargs):
    self._add_mashape_header(kwargs)
    return requests.get(*args, **kwargs)

def _bom_post(self, *args, **kwargs):
    self._add_mashape_header(kwargs)
    return requests.post(*args, **kwargs)

def _get_twitter_data(self, user, full_user_object=False):
    try:
        user_timeline = self.twitter_api.user_timeline(
                user,
                include_rts=True,
                count=200,
                )

    except RateLimitError as e:
        e.args = (self._TWITTER_RL_MSG, 'statuses/user_timeline')
        raise e

    if user_timeline:
        user_data = user_timeline[0]['user']
    else:
        user_data = self.twitter_api.get_user(user)
    screen_name = '@' + user_data['screen_name']

    try:
        search = self.twitter_api.search(screen_name, count=100)
    except RateLimitError as e:
        e.args = (self._TWITTER_RL_MSG, 'search/tweets')
```

```
        raise e

    payload = {
        'mentions': search['statuses'],
        'timeline': user_timeline,
        'user': user_data,
    }

    if not full_user_object:
        payload['user'] = {
            'id_str': user_data['id_str'],
            'screen_name': user_data['screen_name'],
        }

    return payload

####################
#  Public methods  #
####################

def bom_api_path(self, method=''):
    return '/'.join([
        self.api_url.rstrip('/'),
        str(self.api_version),
        method,
    ])



def check_account(self, user, full_user_object=False, return_user_data=False):

    payload = self._get_twitter_data(user, full_user_object=full_user_object)

    if not payload['timeline']:
        raise NoTimelineError(payload['user'])

    url = self.bom_api_path('check_account')
    bom_resp = self._bom_post(url, json=payload)
    bom_resp.raise_for_status()
    classification = bom_resp.json()

    if not return_user_data:
        return classification
    else:
        return classification, payload

def check_accounts_in(self, accounts, full_user_object=False,
                      on_error=None, **kwargs):

    sub_instance = self.create_from(self, wait_on_ratelimit=True,
                                    botometer_api_url=self.api_url)
    max_retries = kwargs.get('retries', 3)

    for account in accounts:
        for num_retries in range(max_retries + 1):
            result = None
            try:
                result = sub_instance.check_account(
                    account, full_user_object=full_user_object)
            except (TweepError, NoTimelineError) as e:
                err_msg = '{}: {}'.format(
                    type(e).__name__,
                    getattr(e, 'msg', '') or getattr(e, 'reason', ''),
                    )
                result = {'error': err_msg}
            except (ConnectionError, HTTPError, Timeout) as e:
```

```
                if num_retries >= max_retries:
                    raise
                else:
                    time.sleep(2 ** num_retries)
            except Exception as e:
                if num_retries >= max_retries:
                    if on_error:
                        on_error(account, e)
                    else:
                        raise

            if result is not None:
                yield account, result
                break
```

8 Start Botometer code

```
from DataCollection import botometer, constants
from botometer import NoTimelineError
from requests import ConnectionError, HTTPError, Timeout
from urllib3.exceptions import ReadTimeoutError, ProtocolError, SSLError
import tweepy
import sys
import os
import glob
import csv
import pandas as pd
import smtplib
import random
import time


class BotometerClient:

   def __init__(self, filename, continuing=False):

       self.bot_meter = botometer.Botometer(wait_on_ratelimit=True,
                                            mashape_key=constants.mashape_key,
                                            **constants.botometer_auth)

       self.master_file_name = 'MasterIDs.csv'
       # Store all the ids we get an error on so they aren't checked again
       self.error_ids_file_name = 'ErrorIDs.csv'
       self.unique_ids_file_name = 'UniqueIDs.csv'

       # Time so we can take how long it takes to scrape all these ids
       self.start_time = time.time()

       if filename.startswith('StreamIDs'):
           self.stream_ids_file_name = filename
           self.streaming_file_name = filename.replace('StreamIDs', 'StreamData')
```

```
            self.timeline_file_name = filename.replace('StreamIDs', 'TimelineData')
            self.mentions_file_name = filename.replace('StreamIDs', 'MentionsData')

        elif filename.startswith('StreamData'):
            self.streaming_file_name = filename
            self.stream_ids_file_name = filename.replace('StreamData', 'StreamIDs')
            self.timeline_file_name = filename.replace('StreamData', 'TimelineData')
            self.mentions_file_name = filename.replace('StreamData', 'MentionsData')

        elif filename.startswith('MERGED-StreamData'):
            self.streaming_file_name = filename
            self.stream_ids_file_name = filename.replace('MERGED-StreamData', 'StreamIDs')
            self.timeline_file_name = filename.replace('MERGED-StreamData', 'TimelineData')
            self.mentions_file_name = filename.replace('MERGED-StreamData', 'MentionsData')

        else:
            print('\nERROR: FILE NAME PROVIDED MUST BE FOR STREAMING, MERGED, OR
                   SCRAPED-IDS CSV!!!')
            return

        self.create_master_file()
        self.create_stream_ids_file()
        self.create_error_file()
        self.create_timeline_file()
        self.create_mentions_file()
        self.tweepy_api = constants.api



        if continuing:
            self.df = BotometerClient.get_remaining_ids(self.stream_ids_file_name,
                                                        self.streaming_file_name,
                                                        self.error_ids_file_name)

            self.stream_ids_df =
                BotometerClient.load_stream_ids_df(self.stream_ids_file_name)

        else:
            self.df = BotometerClient.get_all_ids(self.streaming_file_name,
                      self.master_file_name
                      )

        self.error_df = BotometerClient.load_error_ids_df(self.error_ids_file_name)
        self.master_df = BotometerClient.load_master_ids_df(self.master_file_name)

    def start_bot_collection(self):
        # Get botometer scores for every id in the stream
        print('Starting Client....')
        number_of_accounts_to_check = len(self.df)

        for index, row in self.df.iterrows():
            print('On index: ', index, ' out of ', number_of_accounts_to_check)
            print('row: ', row)
            tweet_text = row['status_text']
            tweet_time = row['status_created_at']
            user_id = row['user_id']
            tweet_count = row['stream_tweet_count']

            if user_id not in self.error_df.user_id.values:

                if user_id in self.master_df.user_id.values:
                    print('UserID already in master using existing bot score value!')
                    master_row = self.master_df.loc[self.master_df['user_id'] == user_id]
                    user = BotometerClient.get_user_data_as_dict(master_row)
                    cap = master_row.iloc[0]['cap']
                    bot_score = master_row.iloc[0]['bot_score']
```

```
                self.save_to_stream_ids(user_id,
                                        bot_score,
                                        cap,
                                        tweet_count,
                                        tweet_time,
                                        tweet_text,
                                        user)

            else:
                try:
                    result, payload =
                   self.bot_meter.check_account(user_id,
                                                 full_user_object=True,
                                                 return_user_data=True)

                    cap = result['cap']['universal']
                    bot_score = result['display_scores']['universal']
                    print('cap: ', cap)
                    print('bot score: ', bot_score)

                    if cap > 0.70:
                        self.send_tweet(payload['user']['screen_name'], cap)

                    # Save to Master, Mentions, and Timeline
                    self.save_to_master(user_id, bot_score, cap, tweet_count,
                                        tweet_time, tweet_text, payload['user'])


                    self.save_to_stream_ids(user_id, bot_score, cap, tweet_count,
                                            tweet_time, tweet_text, payload['user'])

                    self.save_to_mentions(payload['mentions'])
                    self.save_to_timeline(payload['timeline'], cap, bot_score)

                except tweepy.TweepError as exc:
                    # Save this user_id so we don't check it again
                    self.save_to_error_ids(user_id)
                    print('Error encountered for ', user_id)
                    print('Error response: ', exc.response)
                    print('Error reason: ', exc.reason)
                    print('Error api code: ', exc.api_code)
                    print('\n')

                except NoTimelineError as err:
                    self.save_to_error_ids(user_id)
                    print('No Timeline error caught: ', err)
                    print('\n')

                except (ConnectionError, HTTPError, Timeout, ReadTimeoutError,
                           ProtocolError, SSLError) as exc:
                    print("New exception: ", exc)
                    time.sleep(120)

        print('\n\n$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$')
        print('Finished! :)')
        time_diff = int(time.time() - self.start_time)
        num_ids = str(len(self.df))
        print('It look {:02d}:{:02d}:{:02d} time to collect ' + num_ids
              + ' bot scores!'.format(time_diff // 3600, (time_diff % 3600 // 60),
               time_diff % 60))

        BotometerClient.send_notification_email()
        return

    def send_tweet(self, user, cap):
```

```
        low_start_options = ['Beep, Beep, I think I found another bot... {0}'.format(user),
                             'R2 says {0}'.format(user),
                             'It looks like {0}'.format(user),
                             'I\'ve calculated that {}'.format(user)
                             ]

        high_start_options = ['I spy a bot... {0}'.format(user),
                              'Danger Will Robinson I\'ve found another political bot
                               {0}'.format(user),
                              'Robot in disguise {0}'.format(user),
                              'Looks like {0} is breaking the first law: A robot may not
                               injure a human being or, '
                              'through inaction, allow a human being to come to harm.
                               {1}'.format(user, user),
                              'I guess {0} doesn\'t know the Zeroth Law: A robot may not
                               harm humanity, or, by '
                              'inaction, allow humanity to come to harm. {1}'.format(user,
                               user)
                              ]

        cap *= 100
        cap = round(cap, 2)

        if cap < 90:
            start = random.choice(low_start_options)
            ending = ' has a botometer score of {0}%, suggesting it could be a bot or bot
                        Assisted. #politicalbots'.format(cap)
        else:
            start = random.choice(high_start_options)
            ending = ' has a botometer score of {0}%, suggesting it is probably a bot.
                      #politicalbots'.format(cap)

        tweet_text = "{0}{1}".format(start, ending)
        self.tweepy_api.update_status(tweet_text)

        return

    def save_to_error_ids(self, user_id):
        error_ids_file = open(self.error_ids_file_name, 'a')
        error_writer = csv.writer(error_ids_file)

        try:
            error_writer.writerow([user_id])

        except Exception as exc:
            print(exc)
            pass

        error_ids_file.close()
        return

    def save_to_stream_ids(self,
                           user_id,
                           bot_score,
                           cap,
                           tweet_count,
                           tweet_time,
                           tweet_text,
                           user_dict):

        # Open the csv file created previously
        file = open(self.stream_ids_file_name, 'a')

        # Create a csv writer
        writer = csv.writer(file)
```

```
    try:
        writer.writerow([user_id,
                         bot_score,
                         cap,
                         tweet_count,
                         tweet_time,
                         tweet_text,
                         user_dict['favourites_count'],
                         user_dict['statuses_count'],
                         user_dict['description'],
                         user_dict['location'],
                         user_dict['created_at'],
                         user_dict['verified'],
                         user_dict['following'],
                         user_dict['url'],
                         user_dict['listed_count'],
                         user_dict['followers_count'],
                         user_dict['default_profile_image'],
                         user_dict['utc_offset'],
                         user_dict['friends_count'],
                         user_dict['default_profile'],
                         user_dict['name'],
                         user_dict['lang'],
                         user_dict['screen_name'],
                         user_dict['geo_enabled'],
                         user_dict['profile_background_color'],
                         user_dict['profile_image_url'],
                         user_dict['time_zone'],
                         user_dict['listed_count']
                         ])

    except Exception as exc:
        print(exc)
        pass

    # Close the csv file
    file.close()
    return

def save_to_master(self,
                   user_id,
                   bot_score,
                   cap,
                   tweet_count,
                   tweet_time,
                   tweet_text,
                   user_dict):

    # Open the csv file created previously
    master_file = open(self.master_file_name, 'a')

    # Create a csv writer
    master_writer = csv.writer(master_file)

    try:
        master_writer.writerow([user_id,
                                bot_score,
                                cap,
                                tweet_count,
                                tweet_time,
                                tweet_text,
                                user_dict['favourites_count'],
                                user_dict['statuses_count'],
                                user_dict['description'],
                                user_dict['location'],
```

```
                                    user_dict['created_at'],
                                    user_dict['verified'],
                                    user_dict['following'],
                                    user_dict['url'],
                                    user_dict['listed_count'],
                                    user_dict['followers_count'],
                                    user_dict['default_profile_image'],
                                    user_dict['utc_offset'],
                                    user_dict['friends_count'],
                                    user_dict['default_profile'],
                                    user_dict['name'],
                                    user_dict['lang'],
                                    user_dict['screen_name'],
                                    user_dict['geo_enabled'],
                                    user_dict['profile_background_color'],
                                    user_dict['profile_image_url'],
                                    user_dict['time_zone'],
                                    user_dict['listed_count']
                                    ])

        except Exception as exc:
            print(exc)
            pass

        # Close the csv file
        master_file.close()
        return


    def save_to_timeline(self, statuses, cap, bot_score):
        # Open the csv file created previously
        timeline_file = open(self.timeline_file_name, 'a')

        # Create a csv writer
        timeline_writer = csv.writer(timeline_file)

        for status in statuses:
            hashtags, mentions, urls = BotometerClient.parse_entities(status['entities'])

            try:
                # Write the tweet's information to the csv file
                timeline_writer.writerow([status['user']['id'],
                                          cap,
                                          bot_score,
                                          status['text'],
                                          status['created_at'],
                                          status['lang'],
                                          status['place'],
                                          status['coordinates'],
                                          status['id'],
                                          status['favorite_count'],
                                          status['retweeted'],
                                          status['source'],
                                          status['favorited'],
                                          status['retweet_count'],
                                          hashtags,
                                          len(status['entities']['hashtags']),
                                          mentions,
                                          len(status['entities']['user_mentions']),
                                          urls,
                                          len(status['entities']['urls']),
                                          status['entities']
                                          ])
            # If some error occurs
            except Exception as exc:
                print(exc)
```

```
            pass

    # Close the csv file
    timeline_file.close()

    # Return nothing
    return

def save_to_mentions(self, statuses):
    # Open the csv file created previously
    mentions_file = open(self.mentions_file_name, 'a')

    # Create a csv writer
    mentions_writer = csv.writer(mentions_file)

    for status in statuses:
        hashtags, mentions, urls = BotometerClient.parse_entities(status['entities'])

        try:
            # Write the mention's information to the csv file
            mentions_writer.writerow([status['text'],
                                      status['created_at'],
                                      status['lang'],
                                      status['place'],
                                      status['coordinates'],
                                      status['id'],
                                      status['favorite_count'],
                                      status['retweeted'],
                                      status['source'],
                                      status['favorited'],
                                      status['retweet_count'],
                                      hashtags,
                                      len(status['entities']['hashtags']),
                                      mentions,
                                      len(status['entities']['user_mentions']),
                                      urls,
                                      len(status['entities']['urls']),
                                      status['entities'],
                                      status['user']['favourites_count'],
                                      status['user']['statuses_count'],
                                      status['user']['description'],
                                      status['user']['location'],
                                      status['user']['id'],
                                      status['user']['created_at'],
                                      status['user']['verified'],
                                      status['user']['following'],
                                      status['user']['url'],
                                      status['user']['listed_count'],
                                      status['user']['followers_count'],
                                      status['user']['default_profile_image'],
                                      status['user']['utc_offset'],
                                      status['user']['friends_count'],
                                      status['user']['default_profile'],
                                      status['user']['name'],
                                      status['user']['lang'],
                                      status['user']['screen_name'],
                                      status['user']['geo_enabled'],
                                      status['user']['profile_background_color'],
                                      status['user']['profile_image_url'],
                                      status['user']['time_zone'],
                                      status['user']['listed_count']
                                      ])

        # If some error occurs
        except Exception as exc:
            print(exc)
```

```
            pass

    # Close the csv file
    mentions_file.close()
    return

def create_error_file(self):
    if os.path.isfile(self.error_ids_file_name):
        print("Error file found")
        return

    else:
        error_file = open(self.error_ids_file_name, 'w')

        try:
            writer = csv.writer(error_file)
            writer.writerow(['user_id'])

        except Exception as exc:
            print(exc)
            pass

        error_file.close()
        return

def create_master_file(self):
    if os.path.isfile(self.master_file_name):
        print('Master ID file found')
        return

    else:
        print('Creating master ID file...')
        csv_file = open(self.master_file_name, "w")

        try:
            writer = csv.writer(csv_file)

            writer.writerow(['user_id',
                             'bot_score',
                             'cap',
                             'tweet_count',
                             'tweet_time',
                             'tweet_text',
                             'user_favourites_count',
                             'user_statuses_count',
                             'user_description',
                             'user_location',
                             'user_created_at',
                             'user_verified',
                             'user_following',
                             'user_url',
                             'user_listed_count',
                             'user_followers_count',
                             'user_default_profile_image',
                             'user_utc_offset',
                             'user_friends_count',
                             'user_default_profile',
                             'user_name',
                             'user_lang',
                             'user_screen_name',
                             'user_geo_enabled',
                             'user_profile_background_color',
                             'user_profile_image_url',
                             'user_time_zone',
                             'user_listed_count'
                             ])
```

```
            except Exception as exc:
                print('Error writing to csv: ', exc)

            return

    def create_stream_ids_file(self):
        if os.path.isfile(self.stream_ids_file_name):
            print('StreamIDs file found!')
            return

        else:
            print('Creating stream ids file...')
            csv_file = open(self.stream_ids_file_name, "w")

            try:
                writer = csv.writer(csv_file)

                writer.writerow(['user_id',
                                 'bot_score',
                                 'cap',
                                 'tweet_count',
                                 'tweet_time',
                                 'tweet_text',
                                 'user_favourites_count',
                                 'user_statuses_count',
                                 'user_description',
                                 'user_location',
                                 'user_created_at',
                                 'user_verified',
                                 'user_following',
                                 'user_url',
                                 'user_listed_count',
                                 'user_followers_count',
                                 'user_default_profile_image',
                                 'user_utc_offset',
                                 'user_friends_count',
                                 'user_default_profile',
                                 'user_name',
                                 'user_lang',
                                 'user_screen_name',
                                 'user_geo_enabled',
                                 'user_profile_background_color',
                                 'user_profile_image_url',
                                 'user_time_zone',
                                 'user_listed_count'
                                 ])

            except Exception as exc:
                print('Error writing to csv: ', exc)

            return

    def create_timeline_file(self):
        if os.path.isfile(self.timeline_file_name):
            print('Timeline file found')
            return

        timeline_file = open(self.timeline_file_name, 'w')
        timeline_writer = csv.writer(timeline_file)

        try:
            # Write timeline header
            timeline_writer.writerow(['user_id',
                                      'user_cap',
                                      'user_bot_score',
```

```
                                    'status_text',
                                    'status_created_at',
                                    'status_lang',
                                    'status_place',
                                    'status_coordinates',
                                    'status_id',
                                    'status_favorite_count',
                                    'status_retweeted',
                                    'status_source',
                                    'status_favorited',
                                    'status_retweet_count',
                                    'status_hashtags',
                                    'status_hashtag_count',
                                    'status_mentions',
                                    'status_mentions_count',
                                    'status_urls',
                                    'status_url_count',
                                    'status_entities'
                                    ])
        except Exception as exc:
            print('Error writing to timeline csv: ', exc)

        return

    def create_mentions_file(self):
        if os.path.isfile(self.mentions_file_name):
            print('Mentions file found')
            return

        mentions_file = open(self.mentions_file_name, 'w')
        mentions_writer = csv.writer(mentions_file)

        try:
            # Write mentions header
            mentions_writer.writerow(['status_text',
                                    'status_created_at',
                                    'status_lang',
                                    'status_place',
                                    'status_coordinates',
                                    'status_id',
                                    'status_favorite_count',
                                    'status_retweeted',
                                    'status_source',
                                    'status_favorited',
                                    'status_retweet_count',
                                    'status_hashtags',
                                    'status_hashtag_count',
                                    'status_mentions',
                                    'status_mentions_count',
                                    'status_urls',
                                    'status_url_count',
                                    'status_entities',
                                    'user_favourites_count',
                                    'user_statuses_count',
                                    'user_description',
                                    'user_location',
                                    'user_id',
                                    'user_created_at',
                                    'user_verified',
                                    'user_following',
                                    'user_url',
                                    'user_listed_count',
                                    'user_followers_count',
                                    'user_default_profile_image',
                                    'user_utc_offset',
                                    'user_friends_count',
```

```
                                    'user_default_profile',
                                    'user_name',
                                    'user_lang',
                                    'user_screen_name',
                                    'user_geo_enabled',
                                    'user_profile_background_color',
                                    'user_profile_image_url',
                                    'user_time_zone',
                                    'user_listed_count'
                                    ])
        except Exception as exc:
            print('Error writing to csv: ', exc)

        return

    ############################
    # Start of static methods #
    ############################

    @staticmethod
    def get_user_data_as_dict(df):
        print(df)
        user_dict = {'favourites_count': df.iloc[0]['user_favourites_count'],
                     'statuses_count': df.iloc[0]['user_statuses_count'],
                     'description': df.iloc[0]['user_description'],
                     'location': df.iloc[0]['user_location'],
                     'created_at': df.iloc[0]['user_created_at'],
                     'verified': df.iloc[0]['user_verified'],
                     'following': df.iloc[0]['user_following'],
                     'url': df.iloc[0]['user_url'],
                     'listed_count': df.iloc[0]['user_listed_count'],
                     'followers_count': df.iloc[0]['user_followers_count'],
                     'default_profile_image': df.iloc[0]['user_default_profile_image'],
                     'utc_offset': df.iloc[0]['user_utc_offset'],
                     'friends_count': df.iloc[0]['user_friends_count'],
                     'default_profile': df.iloc[0]['user_default_profile'],
                     'name': df.iloc[0]['user_name'],
                     'lang': df.iloc[0]['user_lang'],
                     'screen_name': df.iloc[0]['user_screen_name'],
                     'geo_enabled': df.iloc[0]['user_geo_enabled'],
                     'Profile_background_color':
                          df.iloc[0]['user_profile_background_color'],
                     'profile_image_url': df.iloc[0]['user_profile_image_url'],
                     'time_zone': df.iloc[0]['user_time_zone']}

        return user_dict

    ############################################################
    # Load remaining IDs from a previously generated sample #
    ############################################################
    @staticmethod
    def get_remaining_ids(stream_ids_file_name, streaming_file_name, error_ids_filename):
        # Load streamingData from csv
        path = os.path.dirname(os.path.abspath(__file__)) + '/' + streaming_file_name
        df = pd.read_csv(path,
                         header=0,
                         low_memory=False,
                         error_bad_lines=False,
                         lineterminator='\n')

        # Calculate the tweet count for each user id
        df['stream_tweet_count'] = df.groupby('user_id')['user_id'].transform('count')

        # Drop all the columns we don't care about
        column_list = ['status_text', 'status_created_at', 'user_id', 'stream_tweet_count']
        df = df[column_list]
```

```
        original_size = len(df)

        # Drop duplicate ids since we only need to get the user data once
        df = df.drop_duplicates('user_id', keep='last')
        unique_size = len(df)
        print('Out of ', original_size, ' tweets there were ', (original_size -
                unique_size), ' duplicate ID\'s')

        # Drop any rows that are missing the required columns
        df.dropna(subset=['status_text',
                          'status_created_at',
                          'user_id',
                          'stream_tweet_count'])

        print('Dropped', (unique_size - len(df)), 'rows with missing data!')

        # Load sampledIds from csv
        path = os.path.dirname(os.path.abspath(__file__)) + '/' + stream_ids_file_name
        stream_ids_df = pd.read_csv(path,
                                    header=0,
                                    low_memory=False,
                                    error_bad_lines=False,
                                    lineterminator='\n')

        all_stream_ids = stream_ids_df['user_id'].tolist()

        print('Total number of stream IDs already checked: ', len(stream_ids_df))

        df = df[~df['user_id'].isin(all_stream_ids)]
        print('After comparing with stream ids there are ', len(df), ' ids left!')

        # Read in Error IDs and remove any values already created
        path = os.path.dirname(os.path.abspath(__file__)) + '/' + error_ids_filename

        error_df = pd.read_csv(path,
                               header=0,
                               low_memory=False,
                               error_bad_lines=False,
                               lineterminator='\n')

        error_ids = error_df['user_id'].tolist()

        print('ids in error ids: ', len(error_df))

        df = df[~df['user_id'].isin(error_ids)]
        print('After comparing with error ids there are ', len(df), ' ids left!')

        return df

    @staticmethod
    def get_all_ids(streaming_file_name, master_file_name):

        # Load streamingData from csv
        path = os.path.dirname(os.path.abspath(__file__)) + '/' + streaming_file_name
        master_path = os.path.dirname(os.path.abspath(__file__)) + '/' + master_file_name

        df = pd.read_csv(path,
                         header=0,
                         low_memory=False,
                         error_bad_lines=False,
                         lineterminator='\n')

        master_df = pd.read_csv(master_path,
                                header=0,
                                low_memory=False,
                                error_bad_lines=False,
```

```
                            lineterminator='\n')

    # Calculate the tweet count for each user id
    df['stream_tweet_count'] = df.groupby('user_id')['user_id'].transform('count')

    # Drop all the columns we don't care about
    column_list = ['status_text', 'status_created_at', 'user_id', 'stream_tweet_count']
    df = df[column_list]
    original_size = len(df)

    # Drop duplicate ids since we only need to get the user data once
    df = df.drop_duplicates('user_id', keep='last')
    unique_size = len(df)
    print('Out of ', original_size, ' tweets there were ', (original_size -
                    unique_size), ' duplicate ID\'s')

    # Drop all ids that are already in master_df
    master_id_list = master_df.user_id.tolist()
    df = df[~df.user_id.isin(master_id_list)]

    print('Out of ', unique_size, ' there were ', (unique_size - len(df)), ' ids that
            already have scores')
    print('Collecting bot scores for ', len(df), ' new ids')

    # print('Gathering bot scores for ', unique_size, ' user ids!')



    # Drop any rows that are missing the required columns
    df.dropna(subset=['status_text',
                      'status_created_at',
                      'user_id',
                      'stream_tweet_count'])

    print('Dropped', (unique_size - len(df)), 'rows with missing data!')

    return df

#####################################
# Load Data from Streaming CSV File #
#####################################
@staticmethod
def load_master_ids_df(master_file_name):
    # Read in MasterIDs
    path = os.path.dirname(os.path.abspath(__file__)) + '/' + master_file_name
    master_df = pd.read_csv(path,
                            header=0,
                            low_memory=False,
                            error_bad_lines=False,
                            lineterminator='\n')

    return master_df

@staticmethod
def load_error_ids_df(error_ids_file_name):
    # Read in Error IDs
    path = os.path.dirname(os.path.abspath(__file__)) + '/' + error_ids_file_name
    error_df = pd.read_csv(path,
                           header=0,
                           low_memory=False,
                           error_bad_lines=False,
                           lineterminator='\n')

    return error_df
```

```
@staticmethod
def load_stream_ids_df(stream_ids_file_name):
    # Read in stream IDs
    path = os.path.dirname(os.path.abspath(__file__)) + '/' + stream_ids_file_name
    stream_id_df = pd.read_csv(path,
                               header=0,
                               low_memory=False,
                               error_bad_lines=False,
                               lineterminator='\n')

    return stream_id_df

################################
# One function to rule them all #
################################
@staticmethod
def start_mining(file_name):
    print('\nStarting Botometer mining...')

    # Check if the desired csv file exists
    if os.path.isfile(file_name):
        if file_name.startswith('StreamIDs'):
            print('\nFound SampledIDs file. Continuing to mine...')
            client = BotometerClient(file_name, continuing=True)
        else:
            print('\nStreaming data found')
            client = BotometerClient(file_name)

        # Start it up
        client.start_bot_collection()

    else:
        print('Error: requested csv file does not exist!')
        return

@staticmethod
def show_csv_files():
    print("\nI found the following csv files...")

    path = os.path.dirname(os.path.abspath(__file__))
    extension = 'csv'
    os.chdir(path)
    results = [i for i in glob.glob('*.{}'.format(extension))]
    results.sort()

    for result in results:
        print(result)

    return

###################
# Parsing Methods #
###################

@staticmethod
def parse_entities(entities):
    hashtag_key = 'hashtags'
    mentions_key = 'user_mentions'
    url_key = 'urls'

    if hashtag_key in entities:
        hashtag_dict = entities[hashtag_key]
        hashtag_text = BotometerClient.parse_hashtags(hashtag_dict)
    else:
        hashtag_text = ''
```

```
    if mentions_key in entities:
        mentions_dict = entities[mentions_key]
        mentions_text = BotometerClient.parse_mentions(mentions_dict)
    else:
        mentions_text = ''

    if url_key in entities:
        url_dict = entities[url_key]
        url_text = BotometerClient.parse_urls(url_dict)
    else:
        url_text = ''

    return hashtag_text, mentions_text, url_text

@staticmethod
def parse_hashtags(hashtag_dict):
    hashtag_text = ''
    for dictionary in hashtag_dict:
        if 'text' in dictionary:
            if hashtag_text != '':
                hashtag_text += ' ' + dictionary['text']
            else:
                hashtag_text += dictionary['text']

    return hashtag_text

@staticmethod
def parse_mentions(mentions_dict):
    mentions_text = ''
    for dictionary in mentions_dict:
        if 'id_str' in dictionary:
            if mentions_text != '':
                mentions_text += ' ' + dictionary['id_str']
            else:
                mentions_text += dictionary['id_str']

    return mentions_text

@staticmethod
def parse_urls(url_dict):
    url_text = ''
    for dictionary in url_dict:
        if 'url' in dictionary:
            if url_text != '':
                url_text += ' ' + dictionary['url']
            else:
                url_text += dictionary['url']

    return url_text

######################
# Email Notification #
######################
@staticmethod
def send_notification_email():
    # Email myself when the script finishes so I can start on the next set of data
    server = smtplib.SMTP('smtp.gmail.com', 587)
    server.starttls()
    server.login(constants.email_address, constants.password)

    subject = 'Botometer Script'
    text = 'Botometer Script Finished!'
    message = 'Subject: {}\n\n{}'.format(subject, text)
    server.sendmail(constants.email_address, constants.real_email, message)
    server.quit()
```

```
        return


length = len(sys.argv)
if length == 1:
   print('Error: please provide csv file name or type \'showCSVs\' to see the available
files or type help for '
          'more information')
elif length == 2:
   arg = sys.argv[1]
   if arg == 'showCSVs':
       BotometerClient.show_csv_files()
   elif arg == 'help':
       print('Type showCSVs to see a list of the csv files in this directory that can be
passed as a parameter')
       print('Sample call: python3 start_botometer.py
StreamData-#maga-#qanon-#roseanne-20180531-105244.csv')
   else:
       try:
           BotometerClient.start_mining(arg)
       except Exception as e:
           print('Outer exception', e)
           print('Botometer exception caught')
           BotometerClient.send_notification_email()
```