CS 230 Final Project

Russian Trolls on Twitter

Kathryn Swint and Efua Akonor

Table of Contents

Introduction

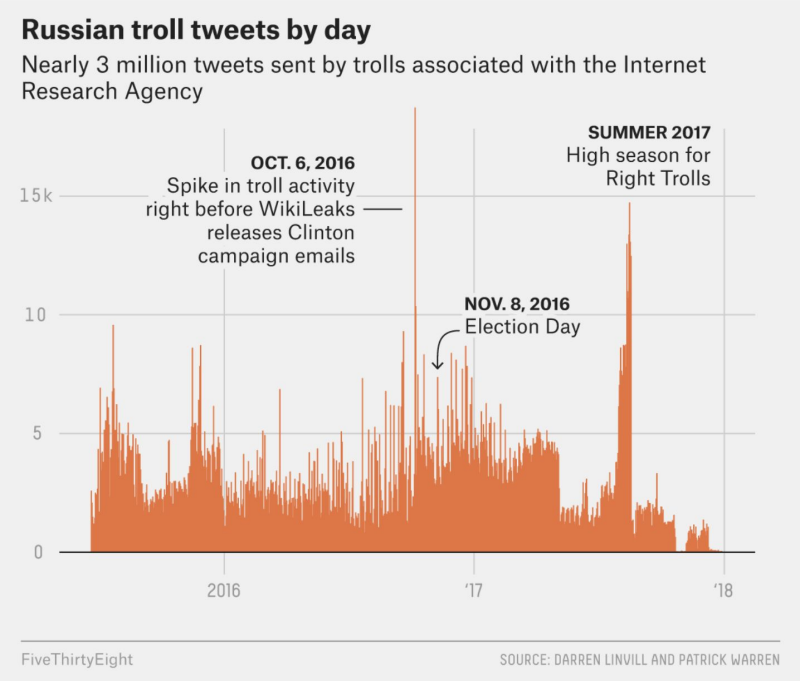
Social media's influence on politics has grown as more and more users spend an increasing amount of time on the platforms, but its influence on the 2016 United States Presidential Election is vastly more complicated than ever before. Troll accounts, defined in this paper as an account or user on a social media platform who posts inflammatory, derogatory, and/or fake information, have been found to be major players in the election of President Donald Trump. In a partially classified report published by the United States Office of the Director of National Intelligence (ODNI) in January 2017, the U.S. Intelligence Community stated that the Russian government (including President Vladimir Putin) directly ordered various Russian agencies—both governmental and commercial—to discredit democratic candidate Hillary Clinton and to promote republican candidate Donald Trump.

The dataset we analyzed includes information about Russian troll accounts (RATs) created and managed by the Internet Research Agency (IRA), a troll farm indirectly funded by the Kremlin. The IRA was the main propagators of online Russian interference in the 2016 election, and had previously been used by the Russian government to spread disinformation about Ukraine. The IRA began promoting President Trump as early as March 2015, 1 year and 8 months before the election itself. The IRA, located in St. Petersburg, Russia, was thought to have over 1,000 employees over the course of Russian interference in the U.S. election.[1]

In July of 2018, FiveThirtyEight published 2,973,371 tweets from 2,848 Twitter accounts identified as being run by the IRA. The tweets were published on GitHub, and include the tweet's author, the author's follower count, the author's following count, and whether or not the tweet was original or a retweet. The earliest tweet was published in February, 2012, and the oldest in May, 2018, though the bulk of the tweets were published between 2015 and 2017. Twitter suspended all RATs identified as originating from the IRA, so the archived tweets (assembled by two Clemson University professors) are the largest assembly of RAT tweets currently available. FiveThirtyEight's article, titled "Why We're Sharing 3 Million Russian Troll Tweets," provides a thorough analysis of the IRA's activity, and we recommend it as an additional resource beyond the brief quotations we provide below.[2]

[1] Seddon, Max. "Documents Show How Russia's Troll Army Hit America." *BuzzFeed News*, BuzzFeed News, 2 June 2014, https://www.buzzfeednews.com/article/maxseddon/documents-show-how-russias-troll-army-hit-america.
[2] Roeder, Oliver. "Why We're Sharing 3 Million Russian Troll Tweets." *FiveThirtyEight*, FiveThirtyEight, 31 July 2018, https://fivethirtyeight.com/features/why-were-sharing-3-million-russian-troll-tweets/.

In the graph below (taken from FiveThirtyEight), the timeline of the tweets and amount of activity shows how the IRA carefully created and responded to major events in the U.S. election.



Source: FiveThirtyEight.com[3]

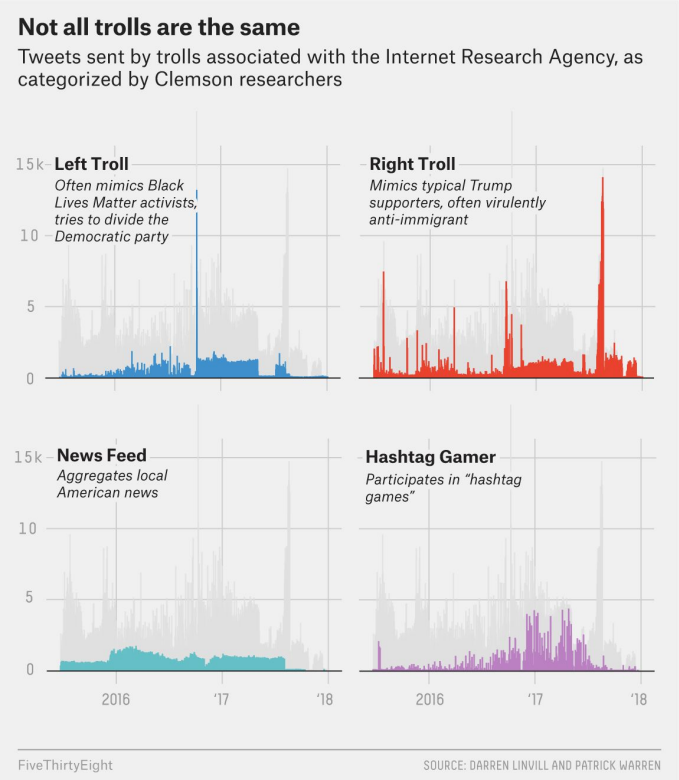
To quote Oliver Roeder's analysis of the graph:

> Even a simple timeline of these tweets can begin to tell a story of how the trolls operated. For instance, there was a flurry of trolling activity on Oct. 6, 2016. As the Washington Post first pointed out using the Clemson researchers' findings, that may have been related to what happened on Oct. 7, 2016, when WikiLeaks released embarrassing emails from the Clinton campaign. There was another big spike in the summer of 2017, when the Internet Research Agency appeared to have shifted its focus to a specific type of troll — one the researchers call the "Right Troll" — that mimicked stereotypical Trump supporters.[4]

The "Right Troll" mentioned above comes from a paper by Darren L. Linvill and Patrick L. Warren, the two Clemson researchers who provided the original data. Linvill and Warren classify the trolls as belonging to one of five groups: Right Troll, Left Troll,

[3] Roeder, "Why We're Sharing 3 Million Russian Troll Tweets."
[4] Roeder, "Why We're Sharing 3 Million Russian Troll Tweets."

News Feed, Hashtag Gamer and Fearmonger.[5] A graph made by FiveThirtyEight charting the group activity of the Left Trolls, Right Trolls, News Feed trolls, and Hashtag Gamer trolls illustrates their activity compared to overall IRA troll activity. To quote from Roeder:



"Right Troll and Left Troll are the meat of the agency's trolling campaign. Right Trolls behave like "bread-and-butter MAGA Americans, only all they do is talk about politics all day long," Linvill said. Left Trolls often adopt the personae of Black Lives Matter activists, typically expressing support for Bernie Sanders and derision for Hillary Clinton, along with "clearly trying to divide the Democratic Party and lower voter turnout." News Feeds are a bit of a mystery: They present themselves as local news aggregators, with names such as @OnlineMemphis and @TodayPittsburgh, and the news they link to is typically legitimate. Hashtag Gamers specialize in playing hashtag games (e.g., #LessInterestingBooks might give rise to the tweet "Waldo's Right Here"); many of their tweets are harmless wordplay in the spirit of the games, but some are socially divisive, in the style of Right Trolls or Left Trolls. And Fearmongers, relatively rare in the data set, spread news about a fake crisis, such as salmonella-contaminated turkeys around Thanksgiving, or the toxic chemical fumes described at the beginning of the New York Times Magazine article about the Internet Research Agency."[6]

It's interesting that many IRA accounts didn't directly spread information and disinformation about the election. Twitter has processes to automatically catch and suspend "bot" accounts, and it seems as though the IRA intentionally worked to create presumed authenticity behind their accounts. As seen in the five categories, creating a network of trolls required creativity and resources that a government attempting to dismantle the security of a G5 country's national elections could provide. Linvill and

[5] Linvill, Darren L, and Patrick L Warren. *Troll Factories: The Internet Research Agency and State-Sponsored Agenda Building*. Clemson University, http://pwarren.people.clemson.edu/Linvill_Warren_TrollFactory.pdf.
[6] Roeder, "Why We're Sharing 3 Million Russian Troll Tweets."

Warren's work to collect, gather, and analyze the work of the IRA has been crucial in understanding the IRA's impact on the 2016 U.S. election. Twitter initially suspended over 36,000 accounts it believed to be associated with Russian internet agencies, but it currently counts 3,841 accounts as having been linked with the IRA.[7] Without work done by newspapers, researchers, and American government agencies, the American public may have never learned of foreign interference in the 2016 U.S. presidential election.

This paper analyzes a subset of the GitHub data using Java programs we created in conjunction with TwitterTrails, an online platform for analyzing the truth of a story via crowd verification and behavior. Run by the TRAILS research team at Wellesley College (including our professor, Takis Metaxas), the platform tracks viral stories on Twitter and how Twitter users interact with disinformation. We were provided a dataset that included 287 trolls (RATs) and 609 stories they tweeted about. We sought to understand trends amongst the RATs and their interaction with the stories.

[7] Calderwood, Alex, et al. "How Americans Wound Up on Twitter's List of Russian Bots." *Wired*, Conde Nast, 29 Nov. 2018, https://www.wired.com/story/how-americans-wound-up-on-twitters-list-of-russian-bots/.

We would like to thank Professor Takis Metaxas, Professor Stella Kakavouli, and Anushe Sheikh for their contributions to this paper.

Methods

Professor Metaxas provided a file of tab-separated values (TSVs), consisting of a user's Twitter handle, their user ID on the TwitterTrails website, the number of tweets they sent relating to stories in the set, the number of stories in the set they tweeted about, and a list of comma-separated values (CSVs) of the stories they tweeted about.

AdjListGraph.java

Designed, written, and tested by Kathryn Swint. AdjListGraph.java is a class within the javafoundations package, used for creating an adjacency list graph. It creates a Vector<T> to store vertices, and a Vector<LinkedList<T>> to store arcs, and the two vectors are its only instance variables. Its methods are listed below, along with descriptions of what they do.

Method	Return Type	Parameters	Description
AdjListsGraph()	void		Constructor. Instantiates the vectors for vertices and arcs.
isEmpty()	boolean		Returns a boolean indicating whether the graph is empty.
getNumVertices()	int		Returns an integer representation of the total number of vertices in the graph.
getNumArcs()	int		Returns an integer representation of the number of arcs.
isArc(T v1, T v2)	boolean	v1: the origin vertex v2: the end vertex	Returns a boolean indicating whether the two vertices are connected. First checks that both vertices exist in the graph, and prints an error message if one or more vertices do not exist.
isEdge(T v1, T v2)	boolean	v1: the origin vertex v2: the end vertex	Returns a boolean indicating whether the two vertices are connected by an edge. First checks that both vertices exist in the graph, and prints an error message if one or more vertices do not exist.
isUndirected()	boolean		Returns a boolean indicating whether the graph is undirected.

addVertex(T v)	void		Adds a vertex to the graph, checking to ensure the vertex doesn't already exist to avoid adding duplicates
removeVertex(T v)	void		Removes a vertex from the graph. Removes the specified vertex from the vertices vector as well as the list of arcs for all vertices that were connected to the vertex
addArc(T v1, T v2)	void	v1: the origin vertex v2: the end vertex	Adds an arc from the origin vertex to the end vertex. First checks that both vertices exist in the graph, and prints an error message if one or more vertices do not exist.
removeArc(T v1, T v2)	void	v1: the origin vertex v2: the end vertex	Removes an arc originating at the origin vertex and ending at the end vertex. First checks that both vertices exist in the graph, and prints an error message if one or more vertices do not exist.
addEdge(T v1, T v2)	void	v1: the first vertex v2: the second vertex	Adds an edge between the two vertices by adding an arc in each direction. First checks that both vertices exist in the graph, and prints an error message if one or more vertices do not exist.
removeEdge(T v1, T v2)	void	v1: the first vertex v2: the second vertex	Removes an edge between the two vertices by removing the arcs in each direction. First checks that both vertices exist in the graph, and prints an error message if one or more vertices do not exist.
getArcs(T v)	LinkedList	v: the vertex whose arcs you'd like to check	Returns a LinkedList of all arcs originating at the specified vertex.
getSuccessors(T v)	LinkedList	v: the vertex whose successors you'd like to check	Returns a LinkedList of the successors of vertex v. Successors are defined as vertices that v is connected to by arcs (with v as the origin) or by edges.
getPredecessors(T v)	LinkedList	v: the vertex whose predecessors you'd like to check	Returns a LinkedList of the predecessors of vertex v. Predecessors are defined as vertices that v is connected to by arcs (with v as the end vertex) or by edges.

saveToTGF(String fileName)	void	fileName: the name of the file you are producing	Produces a .tgf file of the graph. First iterates through and prints all vertices and numbers them, then prints a "#" to adhere to .tgf format, then iterates through and prints all arcs. Uses a PrintWriter().
BFtraversal(T v)	LinkedList	v: the vertex the traversal should begin with	Performs a breadth-first traversal on the graph, beginning at vertex v, and returns a LinkedList of the traversal. Uses three LinkedLists to keep track of the final path, the vertices that have been checked, and the arcs originating from vertex v. Uses a while loop with three nested if statements inside.
DFtraversal(T v)	LinkedList	v: the vertex the traversal should begin with	Performs a depth traversal on the graph, beginning at vertex v, and returns a LinkedList of the traversal. Inspired by the example in our Java Foundations textbook, this method uses an iterator and a stack to perform the traversal.
toString()	String		Returns a nicely formatted string representation of the graph.
main()			Tests the class.

Webpage.java

Designed, written, and tested by Kathryn Swint. Webpage.java was adapted by Kathryn Swint from assignment 5, which was completed with partner Anushe Sheikh. It has three instance variables: URL url to store the URL of the webpage, int numLines to store the number of lines on the page, and String contents to store the HTML contents of the page. We adapted Webpage in order to adapt Cyberspace from PS05 as well, since we wanted a way to automatically pull the titles of stories from the TwitterTrails website.

Method	**Return Type**	**Parameters**	**Description**
Webpage(String u)	void	u: the user-provided URL of the webpage. Given as a string.	Constructor. Instantiates url from String u, then calls readWebpage on u. Has a try/catch to catch MalformedURL exceptions, and prints the exception if it's caught.

readWebpage(String urlName)	void	urlName: the URL of the page we read from	Reads from the webpage one line at a time, instantiates the numLines and content variables in the Webpage object. Uses a scanner, and concatenates each line of HTML into one string. I adapted the method to only grab the title of the RAT story from the webpage. Has a try/catch to catch IO exceptions and IndexOutOfBounds exceptions, and prints the exceptions if they're caught.
compareTo(Webpage w2)	int	w2: the page to compare to	Assumes that if two webpage objects have the same URL, they're the same pages. Returns a positive number (1) if they're identical, and a negative number (-1) if they're different.
getURL()	URL		Returns the URL of the Webpage object.
getNumLines()	int		Returns the numLines of the Webpage object.
getContents()	String		Returns the contents of the Webpage object.
toString()	String		Returns a nicely formatted string displaying a single webpage object.
main()			Tests the class

TwitterTrails.java

Designed, written, and tested by Kathryn Swint. TwitterTrails.java was adapted by Kathryn Swint from Cyberspace.java from assignment 5, which was completed with partner Anushe Sheikh. It has one instance variable: ArrayStack<Webpage> collection to store a stack of Webpage objects. Each story was provided as an ID number that could be used to access its page on the TwitterTrails site. By concatenating the number to the end of the URL "http://twittertrails.wellesley.edu/~trails/stories/investigate.php?id=" it was possible to directly access the page. We chose to build this capability into our project in order to automate accessing the titles of the most and least popular stories. Some of the stories created permission errors when viewing, but our code was able to retrieve the titles for all stories (regardless of permission errors) due to the nature of the HTML on their pages.

Method	Return	Parameters	Description

	Type		
TwitterTrails(Hashtable<String,Integer> hashtable)	Void	hashtable: the hashtable of stories to analyze. Specific to this project.	Constructor for TwitterTrails object. Instantiates collection instance variable, calls method readStories on the hashtable parameter, and prints collection.
readStories(Hashtable<String,Integer> hashtable)	Void	hashtable: the hashtable of stories to analyze. Specific to this project.	Reads from the input file one URL at a time, creates a new Webpage object with the URL it reads, and pushes the new Webpage object into the collection of URLs made by TwitterTrails. Uses an iterator to iterate through the keys in the hashtable, which are the stories' TwitterTrails IDs and can be concatenated onto a standard URL to access their TwitterTrails page.
toString()	String		Returns a formatted string displaying all Webpage objects in the collection.
main()			Tests the class.

RAT.java

Designed, written, and tested by Kathryn Swint. The RAT.java method is used to construct RAT objects. It has five instance variables: String username to represent the user's Twitter handle, String userID to represent the user's TwitterTrails numeric ID, long tweetCount to represent the number of tweets a user published about the stories in the dataset, long storyCount to represent the number of stories a user tweeted about in the dataset, and LinkedList<String> stories to represent the stories that the user tweeted about.

Method	Return Type	Parameters	Description
RAT(String u, String uID, long t, long s)	Void	u: username uID: userID t: tweetCount s: storyCount	Constructor for RAT object. Instantiates all instance variables.
addStory(String story)	Void	story: the story	First checks that the story isn't already in the

		to add to the RAT	RAT's LinkedList of stories, then adds the story if it's new. Has a try/catch to catch NullPointerException, and prints the exception if it's caught.
toString()	String		Returns a formatted string displaying all story objects in the LinkedList of stories.
main()			Tests the class.

RATgraph.java

Designed, written, and tested by Kathryn Swint. Creates an adjacency list graph of RAT objects. We decided to use a separate class for the creation of the RAT graph in order to keep the Investigate class relatively tidy. Capable of exporting the accounts and stories hash tables to a CSV to enable graphing the data in programs like Excel and Google Sheets. RATgraph.java has seven instance variables:

1. AdjListsGraph<String> graph as the adjacency list graph that will hold the RATs and stories
2. Hashtable<String,RAT> accounts to hold the RATs, where the key is the RAT's username and the value is the RAT object
3. Hashtable<String,Integer> stories to hold the stories, where the key is the story ID and the value is the number of times the story has been tweeted about
4. String[] usernames to hold the usernames of the RATs in an interable structure
5. String[] userIDs to hold the user IDs of the RATs in an interable structure
6. int usernamesSize to track the size of the usernames array
7. int userIDsSize to track the size of the userIDs array

It assumes that the user will add usernames or userIDs to their respective arrays, but has no method to remove values from those arrays. We intentionally didn't include those methods since we haven't yet had the need to remove values from the arrays.

Method	**Return Type**	**Parameters**	**Description**
RATgraph(String fileName)	Void	fileName: the name of the file to read	Constructs objects of class RATgraph. Initializes accounts, stories, usernames, usernamesSize, userIDs, userIDsSize, and

		information from	graph. Calls method readFromFile on filename, and createGraph on graph.
readFromFile(String fileName)	Void	fileName: the name of the file to read information from	Reads from a TSV and creates a new RAT object with the information on each line. Uses a scanner to scan through the TSV and create new RAT objects with each line, then uses a for loop to add stories to the newly created RAT object. Uses a try/catch to catch FileNotFoundException and NumberFormatException, and prints the exception if it's caught.
addUsername(String u)	Void	u: the username to add to the usernames array	Adds a username to the usernames array. First checks to see that the array has space. If it doesn't, it expands the size of the array by constructing a temporary array. Also increases the count of usernamesSize by 1.
addUserID(String u)	Void	u: the userID to add to the userIDs array	Adds a userID to the userIDs array. First checks to see that the array has space. If it doesn't, it expands the size of the array by constructing a temporary array. Also increases the count of userIDsSize by 1.
createGraph(String fileName)	Void	fileName: the name of the file to read information from	Creates a graph from the information read by readFromFile. Iterates through the vertices and adds arcs to and from all the stories the user tweeted about. We used arcs instead of edges because it seemed to work better with yED than edges did. Uses a try/catch to catch NullPointerException, and prints the exception and the fileName parameter if it's caught.
exportAccounts(String fileName)	Void	fileName: the name you want your file to have	Iterates through the accounts hashtable to grab the keys, then prints the account's username, storyCount, and tweetCount to a CSV. Uses a try/catch to catch IOException, and prints the exception if it's caught.
exportStories(String fileName)	Void	fileName: the name you want your file to have	Iterates through the stories hashtable to grab the keys, then prints the story's ID and the number of times it was tweeted about to a CSV. Uses a try/catch to catch IOException, and prints the exception if it's caught.
main()			Tests the class.

Investigate.java

Designed, written, and tested by Kathryn Swint, with methods for exporting to a CSV contributed by Efua Akonor. The Investigate.java class has the bulk of our methods for analysis. It has 18 instance variables:

1. RATgraph g: the RATgraph we'll analyze. We decided to use the RATgraph instead of using a TGF for simplicity and to keep Investigate relatively tidy.
2. int highestStories: highest # of stories a RAT participated in
3. int lowestStories: lowest # of stories a RAT participated in
4. int highestRATs: highest # of RATs that participated in one story
5. int lowestRATs: lowest # of RATs that participated in one story
6. int medStoriesParticipated: median # of stories that each RAT participated in
7. int avgStoriesParticipated: average # of stories that each RAT participated in
8. int medRATsPerStory: median # of RATs that participated in each story
9. int avgRATsPerStory: average # of RATs that participated in each story
10. int diameter: the diameter of the graph
11. int radius: the radius of the graph
12. LinkedList<String> centerNodes: the central nodes of the graph
13. int numPops: the number of popular stories the user wants us to analyze, passed as a parameter to the constructor but used in multiple methods
14. Hashtable<String,RAT> mostActiveRATs: a hashtable of the (numPops) most active RATs in the data
15. Hashtable<String,Integer> mostPopStories: a hashtable of the (numPops) most popular stories in the data
16. Hashtable<String,Integer> leastPopStories: a hashtable of the (numPops) least popular stories in the data
17. LinkedList<Integer> storiesPerRATValues: used for calculating medians and averages for how many stories each RAT participated in
18. LinkedList<Integer> RATsPerStoryValues: used for calculating medians and averages for how many RATs participated in each story

Method	Return Type	Parameters	Description
Investigate(String fileName, int	Void	fileName: the name of the file	Constructor. Instantiates instance variables g, numPops, storiesPerRATValues,

numPops)		to read information from numPops: the number of popular stories to analyze	RATsPerStoryValues, centerNodes, mostActiveRATs, mostPopStories, and leastPopStories.
RATsPerStory()	Void		Gets the number of RATs that participated in each story and determines what the highest and lowest number of stories participated in is. Instantiates highestStories and lowestStories instance variables. Adds each value to the RATsPerStoryValues LinkedList.
mostPopularStories(int x)	String	x: how many popular stories to return	Uses an iterator to iterate over the stories hashtable keys, then uses a while loop to compare each stories hashtable value to the highestStories variable. If the value equals the variable, it concatenates that story ID onto a string. If the number of stories it's found is less than x, it then subtracts x from the count and calls itself recursively with the new value. Returns a string representation of the most popular stories in the dataset.
leastPopularStories(int x)	String	x: how many popular stories to return	Uses an iterator to iterate over the stories hashtable keys, then uses a while loop to compare each stories hashtable value to the lowestStories variable. If the value equals the variable, it concatenates that story ID onto a string. If the number of stories it's found is less than x, it then subtracts x from the count and calls itself recursively with the new value. Returns a string representation of the most popular stories in the dataset.
storiesPerRAT()	Void		Gets the number of stories that each RAT participated in and determines what the highest and lowest number of RATs-per-story is. Instantiates highestRATs and lowestRATs instance variables. Adds each value to the storiesPerRATValues LinkedList.
mostActiveRATs(int x)	String	x: how many RATs to return	Uses an iterator to iterate over the accounts hashtable keys, then uses a while loop to compare the number of stories the RAT

			participated in to the highestRATs variable. If the value equals the variable, it concatenates that RAT's username onto a string. If the number of stories it's found is less than x, it then subtracts x from the count and calls itself recursively with the new value. Returns a string representation of the most active RATs in the dataset.
leastActiveRATs(int x)	String	x: how many RATs to return	Uses an iterator to iterate over the accounts hashtable keys, then uses a while loop to compare the number of stories the RAT participated in to the lowestRATs variable. If the value equals the variable, it concatenates that RAT's username onto a string. If the number of stories it's found is less than x, it then subtracts x from the count and calls itself recursively with the new value. Returns a string representation of the least active RATs in the dataset.
normalDistribution(LinkedList<Integer> list)	Void	list: the list to determine the values of. Must either be storiesPerRATValues or RATsPerStoryValues for the method to work properly.	If the list is storiesPerRATValues or RATsPerStoryValues, calculates the average and median values for the list. Then prints one line each with information about the highest number of either RATs per story or stories per RAT, the lowest number, the average number, and the median number (for the same list). Also instantiates the relevant instance variables.If the list is different, prints an error message alerting the user to input one of the valid lists.
getDiameter()	int		Uses breadth-first search to determine the diameter of the graph (where diameter is defined as the largest eccentricity value for any vertex in the graph. Uses a while loop and nested if statements to determine every time the list returned by a breadth-first search switches between printing user vertices and story vertices. Every time a vertex with greater eccentricity is found, the diameter instance variable is updated.
getRadius()	int		Uses breadth-first search to determine the radius of the graph (where radius is defined as the smallest eccentricity value for any vertex

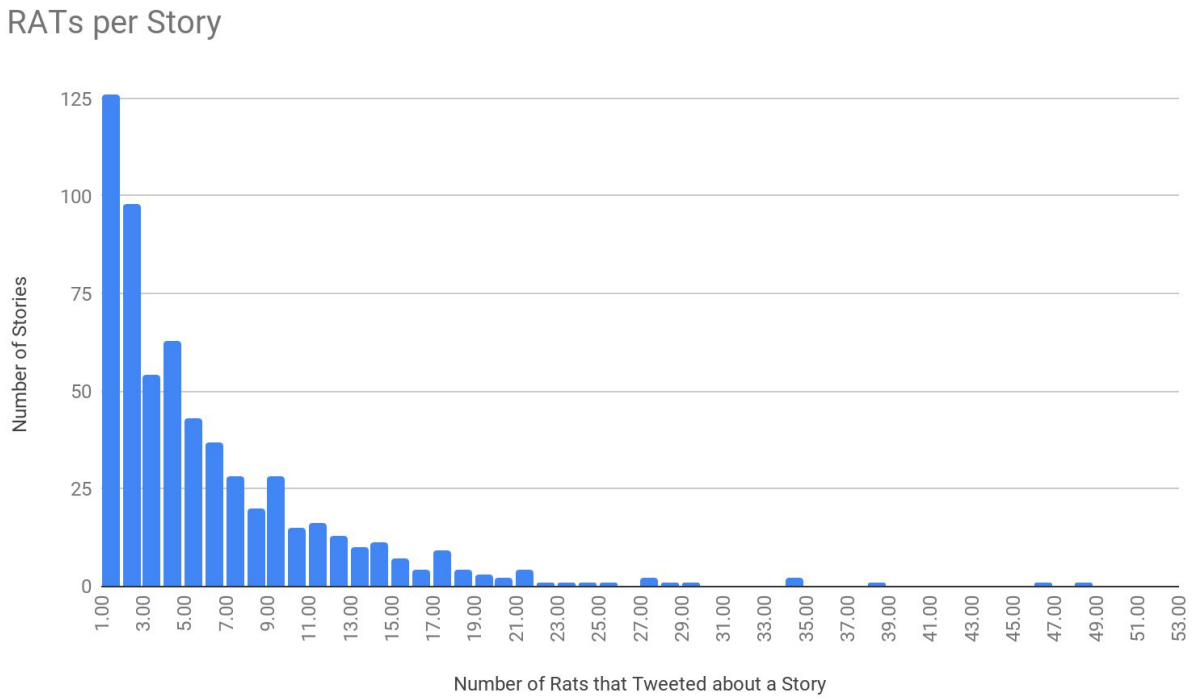
			in the graph. Uses a while loop and nested if statements to determine every time the list returned by a breadth-first search switches between printing user vertices and story vertices. Every time a vertex with smaller eccentricity is found, the radius instance variable is updated. Adds every user that has an eccentricity that equals the radius to the centerNodes list.
isConnected()	boolean		Performs a depth-first search on the first vertex in the graph, then checks that every single account and story is in the list returned by the search. Returns false as soon as a user that is not in the search results is found.
compareRATs(String story1, String story2)	LinkedList< String>	story1: the first story to check story2: the second story to check	Determines how many RATs participated in both stories passed in as parameters. Used to figure out if stories were often tweeted about by the same RATs or if the groups differed. Uses an iterator to iterate over the account keys, checks if the RAT associated with the key tweeted about both stories, and adds the RAT username to a LinkedList if true.
overlappingRATs(Hashtable<String,Integer > hash)	String	hash: the hashtable of stories you want to compare RATs between	Iterates through the keys of the given hashtable of stories. For each key, adds the story ID to the storyIDs array and increments storyIDsSize. Then uses a for loop to iterate through storyIDs and calls compareRATs on the current story and the next story in a nested for loop. Concatenates everything onto a string, and returns the string representation of the common RATs amongst stories in the hash table.
getStoryData()	String		Creates a blank CSV, then iterates through the entire graph of collected story data, collecting the story ID numbers and their number of RAT tweets. Adds the engagement data to a CSV file, which was then used for the bulk of data analysis regarding story engagement. Returns String indicating successful CSV export or any caught errors.
getRatData()	String		Creates a blank CSV, then iterates through the entire graph of collected story data, collecting

			the RAT ID numbers and the number of stories they tweeted. Adds the engagement data to a CSV file, which was then used for the bulk of data analysis regarding RAT participation. Returns String indicating successful CSV export or any caught errors.
main()			Creates a new Investigate object, then calls different methods to analyze and print the answers to questions we have about the dataset.

Conclusions

The Stories

287 RATs participated in the spread of 609 stories in the dataset. There were no identifiable trends in how each RAT interacted with the stories. The number of RATs that tweeted about a single story ranged from 1 to 53, with a median of 5 and an average of 12. Only 19 stories were tweeted about by 20+ RATs, and it was rare to find a “viral story” among the RATs. While the dataset we analyzed only included a fraction of the users that participated in spreading the stories, the disparity between how many users participated in the most and least popular stories was surprising. In the histogram below, the x-axis consists of integers ranging from 1 to 53, and the y-axis shows how many stories were tweeted about by X number of RATs. The heavy clustering towards the left side of the axis shows how infrequently a story was shared by a large number of RATs.



We chose to explore the ten most popular stories in the data set. The table below contains the stories' TwitterTrails ID numbers, the number of RATs that participated in spreading the story, and the title of the story on the Twitter Trails site.

Below, you can see that 6 of the 10 stories are related to President Trump's election and administration. While it seems intuitive that the national politics of the United States would dominate the interest of Russian troll accounts, we're unable to predict whether smaller political stages would mimic the dataset we analyzed for this project.

Story (TwitterTrails ID #)	# of RATs that Participated	Story (TwitterTrails Title)
190816579	53	Mistrial in the first trial of one of the officers accused of killing Freddie Gray
9871042182	46	Trump - 5/20/2016
8281346356	38	Susan Rice
8231029693	34	Kasich to suspend campaign after Indiana primary
4881270967	34	Jeff Sessions confirmation hearing

6781268418	29	White Genocide
9911201671	28	#ifthemediariggedtheelection
4251092648	27	#WorldRefugeeDay
3911263100	27	Pizzagate-2016.12.08
3771181567	25	Megyn Kelly

The next table shows the spread, skepticism, and dates of the first and last tweets about the story (all taken from TwitterTrails). Missing data is due to viewing permissions on the TwitterTrails site.

Story (TwitterTrails ID #)	**Spread**	**Skepticism**	**First Tweet**	**Last Tweet**
190816579	High	Undisputed	April 25, 2015	December 1, 2015
9871042182	Extensive	Undisputed		
8281346356	Extensive	Hesitant	April 2, 2017	April 3, 2017
8231029693				
4881270967	Extensive	Undisputed	November 18, 2016	January 5, 2017
6781268418	High	Hesitant	December 25, 2016	December 25, 2016
9911201671	Moderate	Undisputed	October 19, 2016	October 19, 2016
4251092648				
3911263100	Extensive	Dubious		
3771181567				

Each of the ten most popular stories mostly contained a unique set of users. When comparing the users that participated in one story to users that participated in a second, there wasn't much consistency between the two beyond the reappearance of the most active users in the set. Jenn_Abrams, who was the most active user and tweeted about 144 stories, tweeted about 5 of the 10 most popular stories. NotRitaHart, who was one of

the least active users and tweeted about a single story, didn't tweet about more than 1 top-10 story. interFischer, who tweeted about 5 stories (the median for the data set) tweeted about 2 top-10 stories.

You can see the emphasis on appearing "real" when looking at the ten most popular stories. Story #190816579, the most popular story with 53 RATs tweeting about it, shares 12 RATs with Story #9871042182, the third most popular story with 46 RATs participating. 22.6% of users that participated in the most popular story also participated in the third-most popular story. Strangely enough, only 7 of the ten most active users tweeted about more than one of the ten most popular stories, which we believe reflects how careful the Russian troll accounts were about controlling their collaboration.

The data below shows the users that participated in each of the ten most popular stories. The ten most active users have been underlined as they appear. For comparison, we also checked the overlapping users for the least popular stories. The maximum number of overlaps any of the least popular stories had was 2, but the vast majority of the stories did not have overlapping users.

Below, the bolded headers indicate which story is being compared to the stories listed below (all of which are the ten most popular stories). Each line shows the story's numerical ID, the story's TwitterTrails title, the number of users that "overlapped," and which users "overlapped." Usernames have been hand-censored to exclude slurs, and we apologize if we have missed harmful language in any of the usernames. The ten most active rats have been underlined when they appear.

Story 190816579: Mistrial in the first trial of one of the officers accused of killing — 49 Total, 5.4 Average

Story 4251092648 (2 overlaps): [ImaSwerve, BaoBaeHam]
Story 3771181567 (6 overlaps): [KansasDailyNews, ImaSwerve, AlecMooooody, StanleyParris, FinnaGlo, JavonHIDP]
Story 4881270967 (8 overlaps): [ErRivvvvers, WillisBonnerr, PaulineTT, ImaSwerve, JadonHutchinson, AdrGreerr, melanymelanin, JavonHIDP]
Story 8231029693 (11 overlaps): [ErRivvvvers, KansasDailyNews, PaulineTT, gloed_up, ChadSloyer, WashingtOnline, Jery_RobertsYo, DailySanFran, StanleyParris, OnlineCleveland, iLoveSarahRich]
Story 6781268418 (7 overlaps): [LILJordamn, JerStoner, Jani_s_Jac, BlackNewsOutlet, WatchMeWalkin, RobertEbonyKing, JavonHIDP]
Story 8281346356 (1 overlaps): [ImaSwerve]
Story 9871042182 (12 overlaps): [QueennArielle, Crystal1Johnson, KansasDailyNews, PaulineTT, ChadSloyer, Jani_s_Jac, DetroitDailyNew, JohnnyMarch_, MalloryJared, BlackEyeBlog, JaydaAstonishin, JavonHIDP]
Story 3911263100 (2 overlaps): [WatchMeWalkin, NoJonathonNo]
Story 9911201671 (0 overlaps): []

Story 9871042182: Trump - 5/20/2016 — 56 Total, 6.2 Average

Story 4251092648 (6 overlaps): [DatWiseN***a, redlanews, acejinev, SouthLoneStar, TEN_GOP, rightnpr]
Story 3771181567 (3 overlaps): [KansasDailyNews, SamirGooden, JavonHIDP]
Story 4881270967 (8 overlaps): [DatWiseN***a, JassScott, PaulineTT, BlackToLive, OGDeandre, IIddaaMarks, GwennyThot, JavonHIDP]
Story 8231029693 (15 overlaps): [CannonSher, NewspeakDaily, JohnieVOGUE, DatWiseN***a, DailySanDiego, KansasDailyNews, PaulineTT, KaydenMelton, SamirGooden, TravisRespek, TodayBostonMA, ChadSloyer, Siccerthanyou, TodayPittsburgh, PigeonToday]
Story 6781268418 (7 overlaps): [BrianTheLifter, BlackToLive, Jani_s_Jac, SouthLoneStar, TEN_GOP, GwennyThot, JavonHIDP]
Story 8281346356 (4 overlaps): [OGDeandre, IIddaaMarks, TEN_GOP, GwennyThot]
Story 3911263100 (1 overlaps): [TEN_GOP]
Story 190816579 (12 overlaps): [QueennArielle, Crystal1Johnson, KansasDailyNews, PaulineTT, ChadSloyer, Jani_s_Jac, DetroitDailyNew, JohnnyMarch_, MalloryJared, BlackEyeBlog, JaydaAstonishin, JavonHIDP]

Story 9911201671 (0 overlaps): []

Story 8281346356: Susan Rice — 48 Total, 5.3 Average

Story 4251092648 (3 overlaps): [TheFoundingSon, ImaSwerve, TEN_GOP]
Story 3771181567 (11 overlaps): [GarrettSimpson_, happkendrahappy, ImaSwerve, wadeharriot, PatriotBlake, LeroyLovesUSA, AmelieBaldwin, _NickLuna_, EmileeWaren, CooknCooks, hyddrox]
Story 4881270967 (13 overlaps): [EvaGreen69, happkendrahappy, Aldrich420, ImaSwerve, ChesPlaysChess, JacquelinIsBest, PriceForPierce, OGDeandre, IIddaaMarks, JeffreyKahunas, LauraBaeley, GwennyThot, hyddrox]
Story 8231029693 (0 overlaps): []
Story 6781268418 (3 overlaps): [AmelieBaldwin, TEN_GOP, GwennyThot]
Story 9871042182 (4 overlaps): [OGDeandre, IIddaaMarks, TEN_GOP, GwennyThot]
Story 3911263100 (13 overlaps): [TheFoundingSon, hollandpatrickk, Aldrich420, ChesPlaysChess, PatriotBlake, AmelieBaldwin, WorldnewsPoli, heyits_toby, JeffreyKahunas, TEN_GOP, CooknCooks, LauraBaeley, RealRobert1987]
Story 190816579 (1 overlaps): [ImaSwerve]
Story 9911201671 (0 overlaps): []

Story 8231029693: Kasich to suspend campaign after Indiana primary — 37 Tota, 4.1 Average

Story 4251092648 (3 overlaps): [DatWiseN***a, MrMoraan, Jenn_Abrams]
Story 3771181567 (3 overlaps): [KansasDailyNews, SamirGooden, StanleyParris]
Story 4881270967 (3 overlaps): [ErRivvvvers, DatWiseN***a, PaulineTT]
Story 6781268418 (1 overlaps): [Jenn_Abrams]
Story 8281346356 (0 overlaps): []
Story 9871042182 (15 overlaps): [CannonSher, NewspeakDaily, JohnieVOGUE, DatWiseN***a, DailySanDiego, KansasDailyNews, PaulineTT, KaydenMelton, SamirGooden, TravisRespek, TodayBostonMA, ChadSloyer, Siccerthanyou, TodayPittsburgh, PigeonToday]
Story 3911263100 (0 overlaps): []
Story 190816579 (11 overlaps): [ErRivvvvers, KansasDailyNews, PaulineTT, gloed_up, ChadSloyer, WashingtOnline, Jery_RobertsYo, DailySanFran, StanleyParris, OnlineCleveland, iLoveSarahRich]
Story 9911201671 (1 overlaps): [Jenn_Abrams]

Story 4881270967: Jeff Sessions confirmation hearing — 54 Total, 6 Average

Story 4251092648 (2 overlaps): [DatWiseN***a, ImaSwerve]
Story 3771181567 (5 overlaps): [happkendrahappy, ImaSwerve, finley1589, JavonHIDP, hyddrox]
Story 8231029693 (3 overlaps): [ErRivvvvers, DatWiseN***a, PaulineTT]
Story 6781268418 (5 overlaps): [BlackToLive, DaileyJadon, Jasper_Fly, GwennyThot, JavonHIDP]
Story 8281346356 (13 overlaps): [EvaGreen69, happkendrahappy, Aldrich420, ImaSwerve, ChesPlaysChess, JacquelinIsBest, PriceForPierce, OGDeandre, IIddaaMarks, JeffreyKahunas, LauraBaeley, GwennyThot, hyddrox]
Story 9871042182 (8 overlaps): [DatWiseN***a, JassScott, PaulineTT, BlackToLive, OGDeandre, IIddaaMarks, GwennyThot, JavonHIDP]
Story 3911263100 (6 overlaps): [Aldrich420, ChesPlaysChess, JudeLambertUSA, JeffreyKahunas, LauraBaeley, finley1589]
Story 190816579 (8 overlaps): [ErRivvvvers, WillisBonnerr, PaulineTT, ImaSwerve, JadonHutchinson, AdrGreerr, melanymelanin, JavonHIDP]
Story 9911201671 (4 overlaps): [queenofthewo, MelvinSRoberts, DaileyJadon, AmandaVGreen]

Story 6781268418: White Genocide — 41 Total. 4.5 Average

Story 4251092648 (7 overlaps): [WorldOfHashtags, maymaymyy, GiselleEvns, Pamela_Moore13, SouthLoneStar, TEN_GOP, Jenn_Abrams]
Story 3771181567 (2 overlaps): [AmelieBaldwin, JavonHIDP]
Story 4881270967 (5 overlaps): [BlackToLive, DaileyJadon, Jasper_Fly, GwennyThot, JavonHIDP]
Story 8231029693 (1 overlaps): [Jenn_Abrams]
Story 8281346356 (3 overlaps): [AmelieBaldwin, TEN_GOP, GwennyThot]
Story 9871042182 (7 overlaps): [BrianTheLifter, BlackToLive, Jani_s_Jac, SouthLoneStar, TEN_GOP, GwennyThot, JavonHIDP]
Story 3911263100 (5 overlaps): [DonnaBRivera, AmelieBaldwin, Pamela_Moore13, TEN_GOP, WatchMeWalkin]
Story 190816579 (7 overlaps): [LILJordamn, JerStoner, Jani_s_Jac, BlackNewsOutlet, WatchMeWalkin, RobertEbonyKing, JavonHIDP]
Story 9911201671 (4 overlaps): [KenCannone, DominicValent, DaileyJadon, Jenn_Abrams]

Story 9911201671: #ifthemediariggedtheelection - 10 Total, 1.1 Average
Story 4251092648 (1 overlaps): [Jenn_Abrams]
Story 3771181567 (0 overlaps): []
Story 4881270967 (4 overlaps): [queenofthewo, MelvinSRoberts, DaileyJadon, AmandaVGreen]
Story 8231029693 (1 overlaps): [Jenn_Abrams]
Story 6781268418 (4 overlaps): [KenCannone, DominicValent, DaileyJadon, Jenn_Abrams]
Story 8281346356 (0 overlaps): []
Story 9871042182 (0 overlaps): []
Story 3911263100 (0 overlaps): []
Story 190816579 (0 overlaps): []

Story 4251092648: #WorldRefugeeDay — 30 Total, 3.3 Average
Story 3771181567 (2 overlaps): [KenzDonovan, ImaSwerve]
Story 4881270967 (2 overlaps): [DatWiseN***a, ImaSwerve]
Story 8231029693 (3 overlaps): [DatWiseN***a, MrMoraan, Jenn_Abrams]
Story 6781268418 (7 overlaps): [WorldOfHashtags, maymaymyy, GiselleEvns, Pamela_Moore13, SouthLoneStar, TEN_GOP, Jenn_Abrams]
Story 8281346356 (3 overlaps): [TheFoundingSon, ImaSwerve, TEN_GOP]
Story 9871042182 (6 overlaps): [DatWiseN***a, redlanews, acejinev, SouthLoneStar, TEN_GOP, rightnpr]
Story 3911263100 (4 overlaps): [TheFoundingSon, KenzDonovan, Pamela_Moore13, TEN_GOP]
Story 190816579 (2 overlaps): [ImaSwerve, BaoBaeHam]
Story 9911201671 (1 overlaps): [Jenn_Abrams]

Story 3911263100: Pizzagate-2016.12.08 — 40 Total, 4.4 Average
Story 4251092648 (4 overlaps): [TheFoundingSon, KenzDonovan, Pamela_Moore13, TEN_GOP]
Story 3771181567 (9 overlaps): [DorothieBell, CynthiaMHunter, KenzDonovan, PatriotBlake, AmelieBaldwin, cassishere, CarrieThornthon, CooknCooks, finley1589]
Story 4881270967 (6 overlaps): [Aldrich420, ChesPlaysChess, JudeLambertUSA, JeffreyKahunas, LauraBaeley, finley1589]
Story 8231029693 (0 overlaps): []
Story 6781268418 (5 overlaps): [DonnaBRivera, AmelieBaldwin, Pamela_Moore13, TEN_GOP, WatchMeWalkin]
Story 8281346356 (13 overlaps): [TheFoundingSon, hollandpatrickk, Aldrich420, ChesPlaysChess, PatriotBlake, AmelieBaldwin, WorldnewsPoli, heyits_toby, JeffreyKahunas, TEN_GOP, CooknCooks, LauraBaeley, RealRobert1987]
Story 9871042182 (1 overlaps): [TEN_GOP]
Story 190816579 (2 overlaps): [WatchMeWalkin, NoJonathonNo]
Story 9911201671 (0 overlaps): []

Story 3771181567: Megyn Kelly — 41 Total, 4.5 Average
Story 4251092648 (2 overlaps): [KenzDonovan, ImaSwerve]
Story 4881270967 (5 overlaps): [happkendrahappy, ImaSwerve, finley1589, JavonHIDP, hyddrox]
Story 8231029693 (3 overlaps): [KansasDailyNews, SamirGooden, StanleyParris]
Story 6781268418 (2 overlaps): [AmelieBaldwin, JavonHIDP]
Story 8281346356 (11 overlaps): [GarrettSimpson_, happkendrahappy, ImaSwerve, wadeharriot, PatriotBlake, LeroyLovesUSA, AmelieBaldwin, _NickLuna_, EmileeWaren, CooknCooks, hyddrox]
Story 9871042182 (3 overlaps): [KansasDailyNews, SamirGooden, JavonHIDP]
Story 3911263100 (9 overlaps): [DorothieBell, CynthiaMHunter, KenzDonovan, PatriotBlake, AmelieBaldwin, cassishere, CarrieThornthon, CooknCooks, finley1589]
Story 190816579 (6 overlaps): [KansasDailyNews, ImaSwerve, AlecMooooody, StanleyParris, FinnaGlo, JavonHIDP]
Story 9911201671 (0 overlaps): []

The Russian trolls' tendency to avoid all sharing a single story are reflected in the less popular stories as well. 126 (20.68% of the total) of the stories were only tweeted about once by a RAT in the dataset. With a median of 5 RATs per story, these stories were

able to gain notable virality *without* the Russian trolls making up the bulk of the users who shared them.

Stories with only 1 RAT participating:

961283756, 5521432613, 9851093106, 348650369, 280326762, 7441342406,
3331256708, 567271687, 152707584, 448799713, 664452594, 6441359877,
4601030579, 636979625, 945573148, 186981519, 339456856, 2551367700,
3971000153, 3491122503, 3111206444, 151421537, 681395407, 675737534,
4471005276, 802600700, 948790859, 8981294705, 930507189, 6891377835,
4371051804, 9591048621, 1601395340, 495545783, 1911447200, 853929582,
265510982, 1021427766, 3011208818, 295811690, 100944585, 447824655,
128136504, 945955552, 3721370822, 6621148714, 913793714, 120362941,
5931373766, 160715595, 5381054553, 2261001836, 2581424279, 6351259659,
400583401, 221716833, 2981138236, 964438945, 4071205491, 688495938,
670859395, 3971115987, 421454563, 327757258, 207610726, 924795372,
431468185, 3711281358, 191888184, 3051429728, 712560884, 5461147179,
6031130807, 4371464537, 453904668, 304923861, 4651318812, 345574369,
8241039697, 493734679, 5711242160, 988250809, 774903726, 2971404233,
1211172585, 8201293404, 4001003439, 114549713, 7081398170, 7391284174,
3341083151, 284282902, 395211129, 992943858, 1031074455, 2451384510,
671575624, 882313823, 6201354378, 8661246191, 656554476, 565837237,
7321319452, 6691140623, 824657640, 536917827, 4621002239, 923508292,
3251018569, 3611332348, 6191210916, 314857812, 2701461461, 141613947,
715930790, 2961187461, 716946518, 258521452, 7451070859, 468768274,
8781320799, 6661347918, 412459659, 9241111820, 868972335, 835766384

The table below includes data for 10 randomly selected stories who only had a single RAT share them. The spread generally mimics the 10 most popular stories because TwitterTrails usually focuses on more viral stories, and we only have partial data for those who participated in them. The skepticism ranges from dubious to undisputed, and the date ranges for the timeline of the story are as short as a single day to as long as a couple months. These stories are also slightly less Trump-focused, with only 4 out of 10 stories relating to the 2016 election or Trump.

Story (TwitterTrails ID #)	**Story (TwitterTrails Title)**	**Spread**	**Skepticism**	**First Tweet**	**Last Tweet**
7441342406	Claim: United Airlines did not allow passengers with leggings	High	Undisputed	March 26, 2017	March 26, 2017

	to board the plane				
636979625	Events & Memes: Ted Cruz	Extensive	Undisputed		
8981294705	RoguePOTUSStaff	Moderate	Hesitant	January 26, 2017	January 26, 2017
945955552	#CruzSexScandal	High	Hesitant	March 24, 2016	March 24, 2016
447824655	Claim: Ted Cruz hosting a "Netflix and Chill" event	Insignificant	Dubious	December 28, 2015	December 28, 2015
3971115987	Claim: Bernie will endorse Hillary at an event in NH on 7/12	Moderate	Hesitant	July 7, 2016	July 7, 2016
7391284174	#WomensMarch in Boston	Moderate	Undisputed	January 21, 2017	January 21, 2017
1601395340	Events & Memes: Different Interpretations of Clapper's interview on Russia	Moderate	Dubious	March 2, 2017	May 20, 2017
3611332348	Claim: Rex Tillerson used alias "Wayne Tracker" to discuss climate change while at Exxon	Moderate	Undisputed	March 13, 2017	March 13, 2017
9591048621	Events & Memes: #BernieTrumpDebate	High	Undisputed	May 26, 2016	May 26, 2016

The RATs

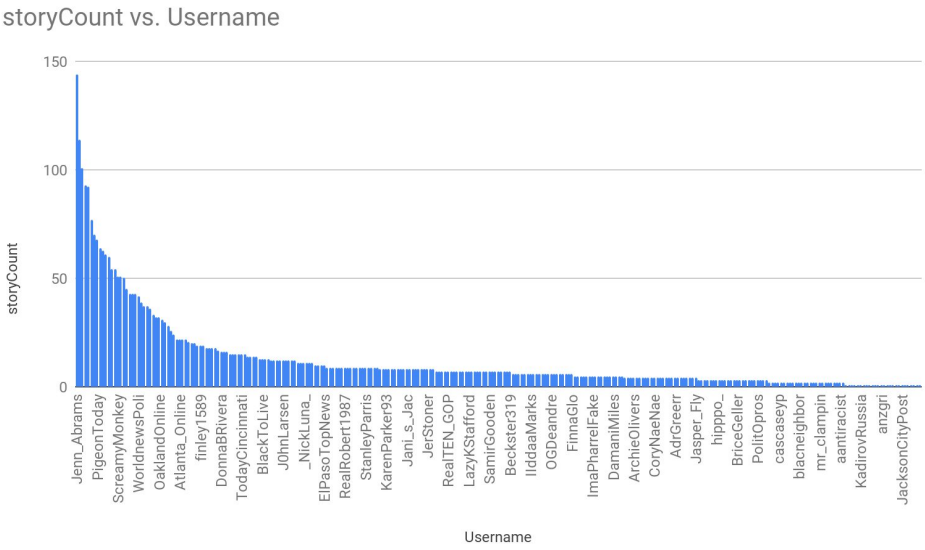
The given dataset contained information about 207 RAT accounts. Much like the stories, the amount of activity coming from each RAT varies greatly, with a large percentage of the RATs active on a small number of stories. The median is 3 and the average is 5. The highest number of stories tweeted about by a single RAT is 144, while the lowest is 1.

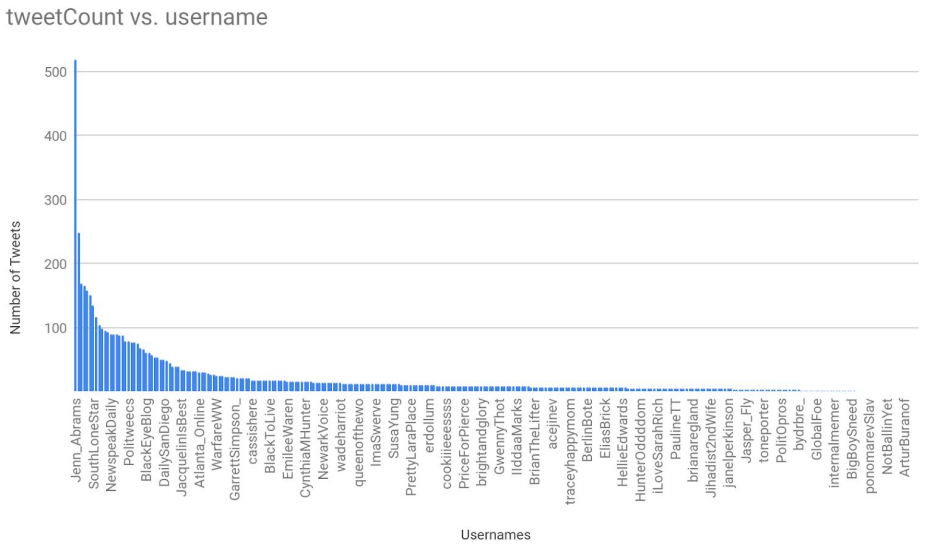
We focused the bulk of our analysis on the ten most active RATs, defining "activity" by how many stories the RAT participated in. Of the ten most popular RATs, only 2 were created to resemble individual users, and both were meant to be women. The other 8 were all created to seem like news outlets for various cities. The most active RATs participated in a minimum of 63 stories, meaning that they participated in 10.4-23.6% of stories in the set. There is some correlation between the tweetCount (the number of tweets a user posted about any story in the dataset) and the storyCount (the number of stories in the dataset the user participated in) in that the ten users with the highest storyCount and the ten users with the highest tweetCount are the same accounts.

Below, you'll see information about the ten most active RATs in the dataset. The dataset included their username, storyCount, and tweetCount, but the following, followers, and total number of tweets values were taken from screenshots (included below) of the accounts before they were deleted.

Twitter username	Given storyCount	Given tweetCount	Following	Followers	Total Number of Tweets
Jenn_Abrams	144	248	22,874	70,083	24,744
KansasDailyNews	114	165	4,980	25,710	50,976
TEN_GOP	101	519	73,704	142,567	10,679
TodayNYCity	93	151	12,857	59,663	48,306
NewOrleansON	92	169	10,424	27,750	45,479
DailySanFran	77	99	4,525	15,741	28,993
WashingtOnline	70	95	12,561	32,134	32,175
PigeonToday	68	91	26,917	29,674	15,464
AmelieBaldwin	64	158	2,285	2,849	39,157
ChicagoDailyNew	63	75	13,064	21,123	48,346

When graphing the storyCount and tweetCount of each user in the dataset, you can clearly see how few stories the majority of users participated in.





As we hypothesized with the stories, we think that the IRA wanted to avoid many of its users interacting with a single story to ensure that Twitter (and real users) would continue assuming the accounts were run by real people. It would also make sense to populate the users' Twitter feeds with tweets not related to Russia nor American politics in order to lend credibility to the users, since the typical American is unlikely to exclusively tweet about politics. Using the Wayback Machine, you can see that most "News Feed" trolls tweeted about the city they claimed to be from and general

international news, while "Right Troll" accounts like Jenn_Abrams spent plenty of time talking about the usual woes of an American millennial.

Below are brief descriptions of information available online about the ten most popular users, including screenshots of their Twitter profiles before they were removed by Twitter. Notice that almost all the accounts were created in 2014.

Jenn_Abrams

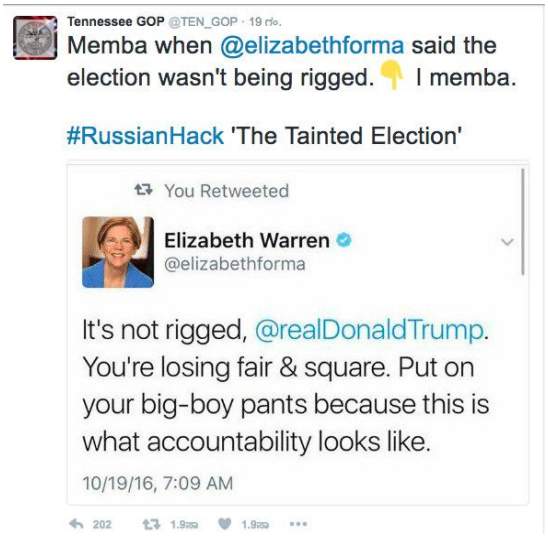
Categorized as a "Right Troll" by the website The Russia Tweets, Jenn_Abrams was the most active user in our dataset. Her account was created long before the IRA began interfering in the U.S. presidential election, and her photo and biography make her seem like any typical Republican voter. The inclusion of the gmail address lends authenticity to the account, while the "location" and "language" sections only further endorse her as a real American Twitter user. The high number of users the account follows would be the only anecdotal indication that the account is different from the typical Twitter user.

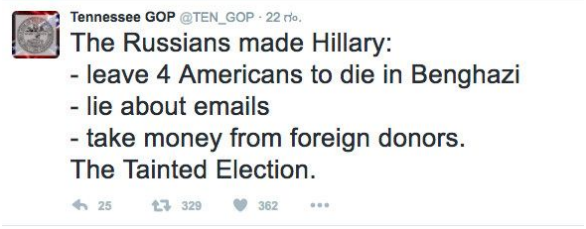
KansasDailyNews

KansasDailyNews falls into the "News Feed" category of users. It has a low ratio of following to followers and a high number of tweets. It lists itself as being located in Kansas City, MO, and its icon mimics local news channel icons.

TEN_GOP

Interestingly, a web search of "TEN_GOP" yields articles across various news outlets' websites about the account. The account was mentioned directly in Special Counsel Robert Mueller's report on Russian interference in the election. A Vox.com article includes screenshots (which have been included below) of Tweets they captured using the Wayback Machine. Their tweets explicitly claim that Russia did not meddle in the election, common Republican messaging about Secretary Clinton, and that Julian Assange (founder of WikiLeaks, the organization that published the hacked DNC emails) "is a hero."





TodayNYCity

TodayNYCity is another "News Feed" account. It has a high number of followers compared to the other accounts, and its logo makes it seem like an independently run media agency. It's unclear whether the account is claiming to be an online-only account or just the Twitter account for an existing media agency. The claim to support international news may provide rationale to users for any tweets about Russia the account shared.

NewOrleansON

NewOrleansON is another "News Feed" account. It has a relatively low number of followers amongst the News Feed accounts, but its 45,479 tweets make it one of the more active RAT accounts in the dataset. Its icon doesn't lend it much credibility, and its name claims its an online-only news source. It almost identically matches the WashingtOnline account.

DailySanFran

The DailySanFran is yet another “News Feed” account. It has a comparatively miniscule number of followers and accounts it follows.

WashingtOnline

With the same logo and biography as the NewOrleansOn account, the two obviously originate from the same source when directly compared. Users may not have noticed the similarities, however, because of the geographic distance between Washington and Louisiana. The IRA may have anticipated the user bases to be different enough to not necessitate changing the visuals on the accounts.

PigeonToday

With a logo that strongly resembles that of Fox News, PigeonToday is a “News Feed” account. It calls itself “America’s Weakest Primetime Lineup” and writes “Merica” instead of “American,” maybe trying to indicate that it’s a humor channel. The mention of

memes and the “coo, man, coo” tagline also indicate that the account was supposed to have a satirical edge.

AmelieBaldwin

The second “Right Troll” account, Amelie Baldwin employs similar tactics as Jenn_Abrams. A photo of a smiling woman and a biography that reads “Wife, Mother, Patriot, Friend” in conjunction with a low follower count makes this account seem like it’s run by a real person.

ChicagoDailyNew

The ChicagoDailyNew account, a “News Feed” troll, is visually identical to the KansasDailyNews account. With the same icon and biography, the accounts (much like the WashingtOnline and NewOrleansON accounts) were probably expected to attract different users who wouldn’t notice their similarities.

While the ten most active users provide lots of data to analyze, the other RATs and stories shouldn't be ignored. The entire graph had a diameter of 8 and a radius of 4, but interestingly the most central nodes were all RATs. The central nodes were comprised of RATs JohnieVOGUE, WillisBonnerr, PaulineTT, gloed_up, PatriotBlake, BlackToLive, Jery_RobertsYo, PamBLMDaniels, AmelieBaldwin, camosaseko, JacquelinIsBest, PigeonToday, SouthLoneStar, JeffreyKahunas, OnlineCleveland, CooknCooks, BlackEyeBlog, LauraBaeley, and RichmondVoice. Of the 19 central nodes (9.17% of the total), 2 of them were one of the ten most active users (AmelieBaldwin and PigeonToday), while all except PamBLMDaniels, camosaseko, and RichmondVoice tweeted about at least two of the ten most popular stories. It makes sense that the central notes were closely linked to the most popular stories, since it brings them closer in proximity to more users and likely lowers their eccentricity.

In comparison to the ten most active users, there were 26 users who only tweeted about one story in the dataset. None of their usernames indicate that they were "News Feed" trolls, while at least 7 either explicitly mention Russia or have vaguely Slavic words in them. One username, "PeeOnHillary," is explicitly referencing a scandal regarding President Trump's use of prostitutes while visiting Russia, while other accounts seem to use generic Anglo-Saxon names.

The least active RATs, all of whom only tweeted about one story: KadirovRussia, Sergeev_Tolik, BigBoySneed, toneporter, CharlesRespeck, LysikVinkova, Adrienne_GG, NotRitaHart, ponomarevSlav, Bocharnikov_V, anzgri, NehemiahX, JmsCoxxx, Kotovamarys, NotBallinYet, abigailssilk, claudia42kern, JacksonCityPost, CalebPaar, PeeOnHillary, SincerePruitt, ArturBuranof, HellieEdwards, RUS_IN_USA, VasilichVasili, dontshootcom

Additional Questions

One additional focus to consider are trends within a larger RAT sample, as well as a sample of engagement from within the general Twitter population. Considering that the most popular story was only shared by 18.46% of our RATs, we'd be excited to access a larger data set to see what the typical percentage of users in a community is required to achieve virality on Twitter. It's important to remember that this engagement is almost entirely artificial, and we assume that a single conglomerate coordinating Twitter engagement likely behaves differently than "real users" do on the same platform. It would be unsurprising were these accounts coordinated to amplify each other while obscuring their offline connection. Before declaring that a political tweet requires 18.46% of a community to reach virality, it would be crucial to compare these data to the general Twitter population.

It would also be interesting to spend time discovering what makes a network of trolls seem organic to a "real" Twitter user. It is incorrect to assume that most Twitter users interact with a random set of other users on the platform, but it's also interesting to see that the Russian trolls seemed to spread their accounts across a multitude of stories. It may raise suspicions if a specific set of accounts consistently retweet each other, but a larger network of accounts that interact indirectly (as this network did) likely mimics "real" Twitter networks.

We also want to investigate how factors other than spread and popularity affect Russian troll engagement. Some factors of interest include the time since the release of a story, and the duration between a stories release and when Russian trolls begin to engage. With more data, it would be interesting to see at what point in a story's timeline Russian trolls were involved, and whether there was some consistency over when they appeared. Duration also seems to affect the popularity of a single story. It's notable that most of the least popular stories mostly occurred over the course of 24 hours. Part of that is probably due to the nature of Twitter and a short news cycle, while another part is probably because a story that can last a week or a month is likely to be tweeted about more simply because users are engaging with it longer.

Code

AdjListGraph.java

```
1. /**
2.  * Creates an adjacency list graph.
3.  *
4.  * @author Kathryn Swint
5.  * @version 12/04/2019
6.  */
7. package javafoundations;
8.
9. import java.util.*;
10. import java.io.*;
11. import java.lang.StackOverflowError;
12. import java.text.Format;
13. import javafoundations.exceptions.*;
14. import java.lang.reflect.Array;
15.
16. public class AdjListsGraph<T> implements Graph<T>{
17.     // instance variables
18.     protected Vector<T> vertices;
19.     protected Vector<LinkedList<T>> arcs;
20.
21.     /**
22.      * Constructor for objects of class AdjListsGraph
23.      */
24.     public AdjListsGraph()
25.     {
26.         vertices = new Vector<T>();
27.         arcs = new Vector<LinkedList<T>>();
28.     }
29.
30.     /**
31.      * Determines whether a graph is empty
32.      *
33.      * @return a boolean indicating whether the graph is empty
34.      */
35.     public boolean isEmpty()
36.     {
37.         return vertices.size() == 0;
38.     }
39.
40.     /**
41.      * Checks the number of vertices in the graph.
42.      *
43.      * @return an integer representation of the number of vertices
44.      */
45.     public int getNumVertices() {
46.         return vertices.size();
```

```
47.     }
48.
49.     /**
50.      * Checks the total number of arcs in the graph.
51.      *
52.      * @return an integer representation of the number of arcs
53.      */
54.     public int getNumArcs(){
55.         int total = 0;
56.         for (int i = 0; i < arcs.size(); i++) {
57.             total += arcs.get(i).size();
58.         }
59.         return total;
60.     }
61.
62.     /**
63.      * Determines whether two vertices are connected by an arc.
64.      *
65.      * @param v1 the first vertex
66.      * @param v2 the second vertex
67.      * @returns boolean indicating whether the two vertices are connected
68.      */
69.     public boolean isArc(T v1, T v2){
70.         boolean connected;
71.         if (vertices.contains(v1) && vertices.contains(v2)) {
72.             int origin = vertices.indexOf(v1); //beginning vertex
73.             int destination = vertices.indexOf(v2); //vertex to connect to
74.             connected = arcs.get(origin).contains(v2); //checks if they're
    connected
75.         } else {
76.             connected = false;
77.             System.out.println("Tried to add edge to one or more vertex that
    doesn't exist.");
78.         }
79.         return connected;
80.     }
81.
82.     /**
83.      * Determines whether two vertices are connected by an edge.
84.      *
85.      * @param v1 the first vertex
86.      * @param v2 the second vertex
87.      * @returns boolean indicating whether the two vertices are connected by an
    edge
88.      */
89.     public boolean isEdge(T v1, T v2){
90.         return (isArc(v1, v2) && isArc(v2, v1));
91.     }
92.
93.     /**
```

```
94.      * Determines whether the graph is undirected.
95.      *
96.      * @returns boolean indicating whether the graph is undirected
97.      */
98.     public boolean isUndirected(){
99.         for (int i = 0; i < arcs.size(); i++) {
100.                LinkedList<T> current = arcs.get(i);
101.                for (int j = 0; j < current.size(); j++) {
102.                    // as soon as two vertices are found that are not connected
    by
103.                    // an edge, returns false
104.                    if (!isEdge(vertices.get(i), current.get(j))) {
105.                        return false;
106.                    }
107.                }
108.            }
109.            return true;
110.        }
111.
112.        /**
113.         * Adds a vertex to the graph
114.         *
115.         * @param v the vertex to be added
116.         */
117.        public void addVertex(T v){
118.            if (!vertices.contains(v)) {
119.                vertices.add(v);
120.                LinkedList<T> vertexEdges = new LinkedList<T>();
121.                arcs.add(vertexEdges);
122.            }
123.        }
124.
125.        /**
126.         * Removes a vertex from the graph
127.         *
128.         * @param v the vertex to be removed
129.         */
130.        public void removeVertex(T v){
131.            if (vertices.contains(v)) { //checks that the vertex is in the graph
132.                for (int i = 0; i < vertices.size(); i++) {
133.                    // remove connections to the vertex from all other vertices
134.                    removeArc(v, vertices.get(i));
135.                    removeArc(vertices.get(i), v);
136.                }
137.                int index = vertices.indexOf(v);
138.                vertices.remove(index);
139.                arcs.remove(index);
140.            } else {
141.                System.out.println("Tried to remove vertex that doesn't exist.");
142.            }
```

```
143.        }
144.
145.        /**
146.         * Adds an arc between two vertices
147.         *
148.         * @param v1 the origin vertex
149.         * @param v2 the destination vertex
150.         */
151.        public void addArc(T v1, T v2){
152.            // checks that both vertices exist
153.            if (vertices.contains(v1) && vertices.contains(v2)) {
154.                int origin = vertices.indexOf(v1);
155.                int destination = vertices.indexOf(v2);
156.                // checks if the vertices are already connected in the specified
    direction
157.                if (!arcs.get(origin).contains(v2)){
158.                    arcs.get(origin).add(v2);
159.                }
160.            } else {
161.                System.out.println("Tried to add edge to one or more vertex that
    doesn't exist.");
162.            }
163.        }
164.
165.        /**
166.         * Removes an arc between two vertices
167.         *
168.         * @param v1 the origin vertex
169.         * @param v2 the destination vertex
170.         */
171.        public void removeArc(T v1, T v2){
172.            // checks that both vertices exist
173.            if (vertices.contains(v1) && vertices.contains(v2)) {
174.                int origin = vertices.indexOf(v1);
175.                int destination = vertices.indexOf(v2);
176.                arcs.get(origin).remove(v2);
177.            } else {
178.                System.out.println("Tried to remove edge from one or more vertex
    that doesn't exist.");
179.            }
180.        }
181.
182.        /**
183.         * Adds an edge between two vertices
184.         *
185.         * @param v1 the first vertex
186.         * @param v2 the second vertex
187.         */
188.        public void addEdge(T v1, T v2){
189.            addArc(v1, v2);
```

```
190.            addArc(v2, v1);
191.        }
192.
193.        /**
194.         * Removes an edge between two vertices
195.         *
196.         * @param v1 the first vertex
197.         * @param v2 the second vertex
198.         */
199.        public void removeEdge(T v1, T v2){
200.            removeArc(v1, v2);
201.            removeArc(v2, v1);
202.        }
203.
204.        /**
205.         * Gets two levels of successors for a given vertex
206.         *
207.         * @return a linked list with two levels of arcs
208.         *
209.         * @param v the vertex to check
210.         */
211.        public LinkedList<T> getArcs(T v) {
212.            LinkedList<T> allArcs = arcs.get(vertices.indexOf(v));
213.            LinkedList<T> temp = new LinkedList<T>();
214.            for (T vertex : allArcs) {
215.                temp.addAll(arcs.get(vertices.indexOf(vertex)));
216.            }
217.            return allArcs;
218.        }
219.
220.        /**
221.         * Returns a linked list of the successors of vertex v.
222.         * Only checks the next level down from v.
223.         *
224.         * @return a linked list with the successors of v
225.         * @param v the vertex you want the successors of
226.         */
227.        public LinkedList<T> getSuccessors(T v){
228.            LinkedList<T> successors = arcs.get(vertices.indexOf(v));
229.            return successors;
230.        }
231.
232.        /**
233.         * Returns a linked list of the predecessors of vertex v.
234.         * Only checks the next level up from v.
235.         *
236.         * @return a linked list with the predecessors of v
237.         * @param v the vertex you want the predecessors of
238.         */
239.        public LinkedList<T> getPredecessors(T v){
```

```
240.            LinkedList<T> predecessors = new LinkedList<T>();
241.            LinkedList<T> currentSuccessors;
242.            T currentVertex;
243.
244.            for (int i = 0; i < vertices.size(); i++) {
245.                currentVertex = vertices.get(i);
246.                currentSuccessors = getSuccessors(currentVertex);
247.                if (currentSuccessors.contains(v)){
248.                    predecessors.add(currentVertex);
249.                }
250.            }
251.
252.            return predecessors;
253.        }
254.
255.        /**
256.         * Creates a TGF file with the vertices and arcs of this graph.
257.         *
258.         * @param fileName the name you want the file to be saved with
259.         */
260.        public void saveToTGF(String fileName){
261.            try {
262.                PrintWriter w = new PrintWriter(new File(fileName));
263.                String s = "";
264.                // prints all the vertices in the right format
265.                for (int i = 0; i < vertices.size(); i++) {
266.                    s += (i + 1) + " " + vertices.get(i) + "\n";
267.                }
268.
269.                s += "#";
270.
271.                for (int i = 0; i < arcs.size(); i++) {
272.                    LinkedList<T> current = arcs.get(i);
273.                    for (T vertex : current) {
274.                        if (isEdge(vertices.get(i), vertex)) {
275.                            s += "\n" + i + " " + vertices.indexOf(vertex);
276.                        }
277.                    }
278.                }
279.
280.                w.println(s);
281.                w.close(); //for tidiness!
282.            }
283.            catch (IOException e) {
284.                System.out.println (e);
285.            }
286.        }
287.
288.        /**
289.         * Performs a breadth-first traversal of the graph, beginning at the
```

```
290.         * user-specificed vertex.
291.         *
292.         * @return a linked list with all vertexes visited in the traversal
293.         * @param v the vertex you want to begin your traversal from
294.         */
295.        public LinkedList<T> BFtraversal(T v){
296.            LinkedList<T> path = new LinkedList<T>();
297.            LinkedList<T> checked = new LinkedList<T>();
298.            LinkedList<T> current = arcs.get(vertices.indexOf(v));
299.            LinkedQueue<T> queue = new LinkedQueue<T>();
300.
301.            queue.enqueue(v);
302.
303.            while (queue.size() != 0) {
304.                if (!checked.contains(v)) {
305.                    for (T vertex : arcs.get(vertices.indexOf(v))) {
306.                        queue.enqueue(vertex);
307.                    }
308.                    checked.add(v);
309.                }
310.
311.                T currentVertex = queue.dequeue();
312.
313.                if (!checked.contains(currentVertex)) {
314.                    for (T vertex : arcs.get(vertices.indexOf(currentVertex))) {
315.                        queue.enqueue(vertex);
316.                    }
317.                    checked.add(currentVertex);
318.                }
319.
320.                if (!path.contains(currentVertex)) {
321.                    path.add(currentVertex);
322.                }
323.            }
324.
325.            return path;
326.        }
327.
328.        /**
329.         * Performs a depth-first traversal of the graph, beginning at the
330.         * user-specificed vertex.
331.         *
332.         * @return a linked list with all vertexes visited in the traversal
333.         * @param v the vertex you want to begin your traversal from
334.         */
335.        public LinkedList<T> DFtraversal(T v)
336.        {
337.            int startIndex = vertices.indexOf(v);
338.            T currentVertex;
339.            LinkedStack<T> traversalStack = new LinkedStack<T>();
```

```
340.            ArrayIterator<T> iter = new ArrayIterator<T>();
341.            boolean[] visited = new boolean[vertices.size()];
342.            boolean found;
343.            LinkedList<T> results = new LinkedList<T>();
344.
345.            if (!vertices.contains(v))
346.                return results;
347.
348.            for (int vertexIdx = 0; vertexIdx < vertices.size(); vertexIdx++)
349.                visited[vertexIdx] = false;
350.
351.            traversalStack.push(vertices.get(startIndex));
352.            iter.add(vertices.get(startIndex));
353.            visited[startIndex] = true;
354.
355.            while (!traversalStack.isEmpty()) {
356.                currentVertex = traversalStack.peek();
357.                found = false;
358.                for (int vertexIdx = 0; vertexIdx < vertices.size() && !found;
    vertexIdx++)
359.                    if (isArc((currentVertex),(vertices.get(vertexIdx))) &&
    !visited [vertexIdx]) {
360.                        traversalStack.push(vertices.get(vertexIdx));
361.                        iter.add(vertices.get(vertexIdx));
362.                        visited[vertexIdx] = true;
363.                        found = true;
364.                    }
365.                if (!found && !traversalStack.isEmpty())
366.                    traversalStack.pop();
367.            }
368.
369.            for (T element : iter.toArray()) {
370.                if (element != null) {
371.                    results.add(element);
372.                }
373.            }
374.
375.            return results;
376.        }
377.
378.        /**
379.         * Standard toString method
380.         *
381.         * @return a string representation of the graph
382.         */
383.        public String toString() {
384.            String result = "Vertices:\n" + vertices + "\nEdges:\n";
385.            for (int i = 0; i < vertices.size(); i++) {
386.                result += "from " + vertices.get(i) + ":\t" + arcs.get(i) + "\n";
387.            }
```

```
388.            return result;
389.        }
390.
391.        public static void main(String args[]) {}
392.    }
```

Webpage.java

```
1.  /**
2.   * Creates a single webpage object with a URL, a number of lines
3.   * on the page, and the HTML on the page. Stores up to the first
4.   * 30 characters.
5.   *
6.   * @author Kat Swint, adapted from work done with Anushe Sheikh
7.   * @version 12/04/2019
8.   */
9.
10. import java.util.Scanner;
11. import java.io.*;
12. import java.net.*;
13. import java.lang.Exception;
14. import java.io.BufferedReader;
15. import java.io.IOException;
16. import java.io.InputStreamReader;
17. import java.net.MalformedURLException;
18. import java.net.URL;
19.
20. public class Webpage implements Comparable<Webpage>
21. {
22.     protected URL url;
23.     protected int numLines = 0;
24.     protected String contents;
25.
26.     /**
27.      * Constructor for objects of class Webpage
28.      * @param u url
29.      */
30.     public Webpage(String u)
31.     {
32.         try {
33.             url = new URL(u);
34.             readWebpage(u); // Instantiates numLines and content & prevents
35.             // having to create object and read lines separately.
36.         } catch (MalformedURLException ex){ // makes sure that the url is valid
37.             System.out.println(ex);
38.         }
39.     }
40.
41.     /**
```

```
42.      * Reads from the webpage one line at a time, instantiates the
43.      * numLines and content variables in the Webpage object
44.      *
45.      * @exception IOException thrown when the input is invalid
46.      * @param urlName the URL of the page we read from
47.      */
48.     public void readWebpage(String urlName) {
49.         try {
50.             URL u = new URL(urlName);
51.             Scanner urlScan = new Scanner(u.openStream());
52.             int count = 0;
53.             String allContents = "";
54.             contents = "";
55.
56.             while (urlScan.hasNext()) {
57.                 count += 1;
58.                 allContents += urlScan.nextLine(); //concatenates each line into
    one string
59.             }
60.
61.             numLines = count;
62.             allContents = allContents.substring(465,allContents.length());
63.
64.             for (int i = 0; i < (allContents.length() - 13); i++) {
65.                 if (!allContents.substring(i,(i + 13)).equals("TwitterTrails"))
    {
66.                     contents += allContents.charAt(i);
67.                 } else {
68.                     break;
69.                 }
70.             }
71.
72.             if (contents.substring(contents.length() - 3,
    contents.length()).equals(" - ")) {
73.                 contents = contents.substring(0, contents.length() - 3);
74.             }
75.         } catch (IOException ex){
76.             System.out.println(ex);
77.         } catch (IndexOutOfBoundsException ex) {
78.             System.out.println(ex);
79.         }
80.     }
81.
82.     /**
83.      * Assumes that if two webpage objects have the same URL, they're
84.      * the same pages. Returns a positive number (1) if they're
85.      * identical.
86.      *
87.      * @param w2 the webpage you want to compare it to
88.      */
```

```
89.     public int compareTo(Webpage w2){
90.         URL w2URL = w2.getURL();
91.         if (w2URL.equals(url)) {
92.             return 1;
93.         }
94.         else {
95.             return -1;
96.         }
97.     }
98.
99.     /**
100.         * Getter for URL.
101.         *
102.         * @return the URL
103.         */
104.        public URL getURL() {
105.            return url;
106.        }
107.
108.        /**
109.         * Getter for numLines.
110.         *
111.         * @return numLines
112.         */
113.        public int getNumLines() {
114.            return numLines;
115.        }
116.
117.        /**
118.         * Getter for content.
119.         *
120.         * @return contents
121.         */
122.        public String getContents() {
123.            return contents;
124.        }
125.
126.        /**
127.         * toString method.
128.         *
129.         * @return a formatted string displaying a single webpage object
130.         */
131.        public String toString()
132.        {
133.            String u = url.toString();
134.            int storyLength = u.length() - 70;
135.            return ("Story " + u.substring(u.length() - storyLength, u.length())
    + ": " + contents);
136.        }
137.
```

```
138.        /**
139.         * Main method with testing code
140.         */
141.        public static void main(String[] args) {}
142.    }
```

TwitterTrails.java

```
1.  /**
2.   * Adapted from the Cyberwalk class from a previous PSET.
3.   * "TwitterTrails is used to contain and maintain a collection of Webpage
    objects.
4.   * it provides functionality related to storing, retrieving, and/or printing
5.   * webpages in a LIFO manner, one per line." - Assignment instructions
6.   *
7.   * @author Kat Swint, adapted from work done with Anushe Sheikh
8.   * @version 12/06/2019
9.   */
10.
11. import java.util.Scanner;
12. import java.util.Iterator;
13. import java.io.*;
14. import javafoundations.ArrayStack;
15. import java.util.Hashtable;
16.
17. public class TwitterTrails
18. {
19.     private ArrayStack<Webpage> collection;
20.
21.     /**
22.      * Constructor for objects of class TwitterTrails
23.      *
24.      * @param hashtable the hashtable made in Investigate that we want to
    process
25.      */
26.     public TwitterTrails(Hashtable<String,Integer> hashtable)
27.     {
28.         collection = new ArrayStack<Webpage>();
29.         readStories(hashtable);
30.         System.out.println(collection);
31.     }
32.
33.     /**
34.      * Reads from the input file one URL at a time, creates a new
35.      * Webpage object with the URL it reads, and pushes the new
36.      * Webpage object into the collection of URLs made by TwitterTrails.
37.      *
38.      * @exception FileNotFoundException thrown when the input file
39.      * is not found
```

```
40.      */
41.     public void readStories(Hashtable<String,Integer> hashtable){
42.         Iterator<String> iterator = hashtable.keySet().iterator(); //iterates
    over stories keys
43.
44.         while(iterator.hasNext()) {
45.             String key = iterator.next(); //the next story in the Vector
46.             String url =
    "http://twittertrails.wellesley.edu/~trails/stories/investigate.php?id=" + key;
47.             Webpage w = new Webpage(url);
48.             collection.push(w);
49.         }
50.     }
51.
52.     /**
53.      * toString method.
54.      *
55.      * @return a formatted string displaying a all webpage objects in a
56.      *         collection and the webpage with the most number of lines
57.      */
58.     public String toString(){
59.         ArrayStack<Webpage> tempStack = new ArrayStack<Webpage>(); // creates a
    new tempArray stack to aid printing
60.         String s = "\n";
61.         int mostLines = 0;
62.         String mL = ""; // records which website has the most content on page,
    updated in if-while loop below
63.
64.         if (!collection.isEmpty()) {
65.             // loops through webpage objects to print and compare their content
    lengths
66.             while (!collection.isEmpty()) {
67.                 Webpage page = collection.pop();
68.                 s += page.toString() + "\n";
69.                 tempStack.push(page);
70.                 if (page.numLines > mostLines) { // updates which webpage has
    most number of lines
71.                     mostLines = page.numLines;
72.                     mL = page.toString();
73.                 }
74.             }
75.
76.             // restores the original stack by transferring objects from
    temporary stack
77.             while (!tempStack.isEmpty()) {
78.                 collection.push(tempStack.pop());
79.             }
80.
81.             s += "\nThe largest Webpage was: " + mL;
82.         }
```

```
83.
84.         return s;
85.     }
86.
87.     /**
88.      * Main method with testing code
89.      */
90.     public static void main(String[] args){}
91. }
```

RAT.java

```
1.  /**
2.   * Represents (and constructs) a single RAT object.
3.   *
4.   * @author Kathryn Swint
5.   * @version 12/05/2019
6.   */
7.
8.  import java.util.LinkedList;
9.
10. public class RAT
11. {
12.     // instance variables
13.     protected String username;
14.     protected String userID;
15.     protected long tweetCount;
16.     protected long storyCount;
17.     protected LinkedList<String> stories;
18.
19.     /**
20.      * Constructor for objects of class RAT
21.      *
22.      * @param u the RAT's username
23.      * @param uID the RAT's userID
24.      * @param t the RAT's tweetCount
25.      * @param s the RAT's storyCount
26.      */
27.     public RAT(String u, String uID, long t, long s)
28.     {
29.         this.username = u;
30.         this.userID = uID;
31.         this.tweetCount = t;
32.         this.storyCount = s;
33.         stories = new LinkedList<String>();
34.     }
35.
36.     /**
37.      * Adds a story to the RAT's LinkedList of stories.
```

```
38.      *
39.      * @param story the story to be added
40.      */
41.     protected void addStory(String story) {
42.         try {
43.             if (!stories.contains(story)) {
44.                 stories.add(story); //avoiding adding duplicate stories
45.             }
46.         } catch (NullPointerException ex) {
47.             System.out.println(ex + " for story " + story);
48.         }
49.     }
50.
51.     /**
52.      * Creates a nicely formatted string representation of a RAT object.
53.      *
54.      * @return a string representation of a RAT object
55.      */
56.     public String toString() {
57.         String allStories = "";
58.         for (String s : stories) {
59.             allStories += s + ", ";
60.         }
61.         allStories = allStories.substring(0, allStories.length()-3); //removes
    the last "," and space
62.         return username + "\t" + userID + "\t"
63.         + tweetCount + "\t" + storyCount + "\t" + allStories; //nicely formatted
    string
64.     }
65.
66.     public static void main(String[] args) {
67.     }
68. }
```

RATgraph.java

```
1.  /**
2.   * Creates a graph of RAT objects based on information from a TSV.
3.   *
4.   * @author Kathryn Swint
5.   * @version 12/05/2019
6.   */
7.
8.  import javafoundations.*;
9.  import java.util.*;
10. import java.io.PrintWriter;
11. import java.io.File;
12. import java.io.FileNotFoundException;
13. import java.io.IOException;
```

```
14.
15. public class RATgraph
16. {
17.     // instance variables
18.     protected AdjListsGraph<String> graph; //how we'll graph the RATs and
    stories
19.
20.     protected Hashtable<String,RAT> accounts; //a hashtable of RATs.
21.     //key = username, value = RAT object
22.     protected Hashtable<String,Integer> stories; //a hashtable of stories. key =
    story,
23.     //value = # of times the story was tweeted about
24.     protected String[] usernames; //array of RAT usernames
25.     protected String[] userIDs; //array of RAT userIDs
26.
27.     protected int usernamesSize; //number of usernames in the usernames array
28.     protected int userIDsSize; //number of userIDs in the userIDs array
29.
30.     /**
31.      * Constructor for objects of class RATgraph
32.      *
33.      * @param fileName the name of the file to read from
34.      */
35.     public RATgraph(String fileName)
36.     {
37.         // initialise instance variables
38.         accounts = new Hashtable<String,RAT>();
39.         stories = new Hashtable<String,Integer>();
40.
41.         usernames = new String[300]; //assumes there are <=300 RATs
42.         userIDs = new String[300];
43.
44.         usernamesSize = 0;
45.         userIDsSize = 0;
46.
47.         readFromFile(fileName); //reads RATs from file
48.
49.         graph = new AdjListsGraph<String>();
50.         createGraph(fileName); //creates the graph of RATs
51.     }
52.
53.     /**
54.      * Reads from a TSV and creates a new RAT object with the information on
    each
55.      * line.
56.      *
57.      * @param fileName the file to read from
58.      */
59.     public void readFromFile(String fileName)
60.     {
```

```
61.         try{
62.             Scanner scan = new Scanner(new File(fileName));
63.             String header = scan.nextLine(); //takes care of file header
64.
65.             while (scan.hasNext()) {
66.                 String[] input = scan.nextLine().split("\t"); //splits @ tabs
    since file is a TSV
67.
68.                 String user = input[0];
69.                 String accountNum = input[1];
70.                 long tweetCount = Long.parseLong(input[2]);
71.                 long storyCount = Long.parseLong(input[3]);
72.
73.                 addUsername(user); //adds username to the username array
74.                 addUserID(accountNum); //adds userID to the userID array
75.
76.                 RAT current = new RAT(user, accountNum, tweetCount, storyCount);
77.                 String[] currentStories = input[4].split(","); //splits the
    stories at commas
78.
79.                 for (String s : currentStories) {
80.                     current.addStory(s); //adds the story to the current RAT
81.                     if (!stories.containsKey(s)) {
82.                         stories.put(s, 1); //if the story is not in the stories
    Hashtable,
83.                         //adds it with a "tweeted about" count of 1
84.                     } else {
85.                         stories.replace(s, (1 + stories.get(s))); //increases
    "tweeted about" count
86.                     }
87.                 }
88.
89.                 accounts.put(user, current); //adds this RAT to the hashtable of
    RATs
90.             }
91.
92.             scan.close();
93.         } catch (FileNotFoundException ex) {
94.             System.out.println("File " + fileName + " was not found.");
95.         } catch (NumberFormatException ex) {
96.             System.out.println(ex);
97.         }
98.     }
99.
100.        /**
101.         * Adds a username to the usernames array.
102.         *
103.         * @param u the username to add
104.         */
105.        private void addUsername(String u) {
```

```
106.            if (usernamesSize == usernames.length) {
107.                String[] temporary = new String[usernames.length * 2];
108.
109.                for (int i = 0; i < usernames.length; i++) {
110.                    temporary[i] = usernames[i];
111.                }
112.
113.                usernames = temporary;
114.            }
115.
116.            usernames[usernamesSize++] = u; //adds the username to the array &
    increases usernamesSize
117.        }
118.
119.        /**
120.         * Adds a userID to the userIDs array
121.         *
122.         * @param u the userID to add
123.         */
124.        private void addUserID(String u) {
125.            if (userIDsSize == userIDs.length) {
126.                String[] temporary = new String[userIDs.length * 2];
127.
128.                for (int i = 0; i < userIDs.length; i++) {
129.                    temporary[i] = userIDs[i];
130.                }
131.
132.                userIDs = temporary;
133.            }
134.
135.            userIDs[userIDsSize++] = u; //adds the userID to the array &
    increases userIDsSize
136.        }
137.
138.        /**
139.         * Creates a graph from the accounts and stories Vertex structures.
140.         *
141.         * @param fileName the fileName you read from previously (used here
142.         *          for the exception).
143.         */
144.        protected void createGraph(String fileName)
145.        {
146.            try {
147.                String currentUser;
148.                LinkedList<String> currentStories;
149.                RAT currentRAT;
150.                String currentU;
151.
152.                for (int i = 0; i < usernamesSize; i++) {
153.                    currentU = usernames[i]; //gets username from usernames array
```

```
154.                    currentRAT = accounts.get(currentU); //gets RAT from accounts
    using username as key
155.
156.                    graph.addVertex(currentU); //adds username as a vertex to the
    graph
157.                    for (String story : currentRAT.stories) {
158.                        graph.addVertex(story); //adds the story to the graph
159.
160.                        //used arcs instead of edges here to ensure that yEd
    correctly displayed
161.                        //our bipartite graph. We're unsure why edges created a
    bug in yEd.
162.                        graph.addArc(currentU, story); //creates an arc between
    the user and story
163.                        graph.addArc(story, currentU); //creates an arc between
    the story and use
164.                    }
165.                }
166.            } catch (NullPointerException ex) {
167.                System.out.println(ex + " for file " + fileName);
168.            }
169.        }
170.
171.        public void exportAccounts(String fileName) {
172.            try {
173.                Iterator<String> iterator = accounts.keySet().iterator();
    //iterates over accounts keys
174.                String s = "Username,storyCount,tweetCount\n";
175.
176.                while(iterator.hasNext()) {
177.                    String key = iterator.next();
178.                    RAT current = accounts.get(key);
179.                    s += key + "," + current.storyCount  + "," +
    current.tweetCount + "\n";
180.                }
181.
182.                PrintWriter w = new PrintWriter(new File(fileName));
183.                w.println(s);
184.                w.close(); //for tidiness!
185.            }
186.            catch (IOException e) {
187.                System.out.println (e);
188.            }
189.        }
190.
191.        public void exportStories(String fileName) {
192.            try {
193.                Iterator<String> iterator = stories.keySet().iterator();
    //iterates over accounts keys
194.                String s = "storyID,times_tweeted_about";
```

```
195.
196.                 while(iterator.hasNext()) {
197.                     String key = iterator.next();
198.                     int current = stories.get(key);
199.                     s += key + "," + current + "\n";
200.                 }
201.
202.                 PrintWriter w = new PrintWriter(new File(fileName));
203.                 w.println(s);
204.                 w.close(); //for tidiness!
205.             }
206.             catch (IOException e) {
207.                 System.out.println (e);
208.             }
209.         }
210.
211.         public static void main(String[] args) {}
212.     }
```

Investigate.java

```
1.  /**
2.   * Creates a graph of RAT objects using class RATgraph, then
3.   * analyzes the graph's RAT and story nodes and their connections.
4.   * Produces printed strings as output.
5.   *
6.   * @author Kathryn Swint
7.   * @version 12/05/2019
8.   */
9.
10. import java.util.*;
11. import java.io.*;
12. import javafoundations.*;
13.
14. public class Investigate
15. {
16.     protected RATgraph g;
17.
18.     protected int highestStories; //highest # of stories a RAT participated in
19.     protected int lowestStories; //lowest # of stories a RAT participated in
20.
21.     protected int highestRATs; //highest # of RATs that participated in one
    story
22.     protected int lowestRATs; //lowest # of RATs that participated in one story
23.
24.     protected int medStoriesParticipated; //median # of stories that each RAT
    participated in
```

```
25.     protected int avgStoriesParticipated; //average # of stories that each RAT
    participated in
26.
27.     protected int medRATsPerStory; //median # of RATs that participated in each
    story
28.     protected int avgRATsPerStory; //average # of RATs that participated in each
    story
29.
30.     protected int diameter;
31.     protected int radius;
32.     protected LinkedList<String> centerNodes;
33.
34.     protected int numPops;
35.
36.     protected Hashtable<String,RAT> mostActiveRATs;
37.     protected Hashtable<String,Integer> mostPopStories;
38.     protected Hashtable<String,Integer> leastPopStories;
39.
40.     protected LinkedList<Integer> storiesPerRATValues; //used for calculating
    medians and averages
41.     protected LinkedList<Integer> RATsPerStoryValues; //used for calculating
    medians and averages
42.
43.     /**
44.      * Constructor for objects of class Investigate
45.      *
46.      * @param fileName the file to read from
47.      * @param numPops how many "popular" or "active" stories/RATs to show
48.      */
49.     public Investigate(String fileName, int numPops) {
50.         g = new RATgraph(fileName);
51.
52.         this.numPops = numPops;
53.
54.         storiesPerRATValues = new LinkedList<Integer>();
55.         RATsPerStoryValues = new LinkedList<Integer>();
56.         centerNodes = new LinkedList<String>();
57.         mostActiveRATs = new Hashtable<String,RAT>();
58.         mostPopStories = new Hashtable<String,Integer>();
59.         leastPopStories = new Hashtable<String,Integer>();
60.     }
61.
62.     /**
63.      * Gets the number of RATs that participated in each story and determines
64.      * what the highest and lowest number of stories participated in is.
65.      */
66.     protected void RATsPerStory() {
67.         Iterator<String> iterator = g.stories.keySet().iterator(); //iterates
    over stories keys
68.
```

```
69.         highestStories = 0;
70.         lowestStories = 5000; //assumes that no one participated in as many as
    5000 stories
71.         // (given that there aren't 5000 stories)
72.
73.         while(iterator.hasNext()) {
74.             String key = iterator.next(); //the next story in the Vector
75.             int currentRATs = g.stories.get(key); //the number of RATs that
    participated in
76.             //each story
77.             RATsPerStoryValues.add(currentRATs); //adds the number to a
    LinkedList of values for
78.             //later use
79.             if (currentRATs >= highestStories) { //if the RATs count is higher,
    updates highestStor
80.                 highestStories = currentRATs;
81.             }
82.
83.             if (currentRATs <= lowestStories) { //if the RATs count is lower,
    updates lowestStories
84.                 lowestStories = currentRATs;
85.             }
86.         }
87.     }
88.
89.     /**
90.      * Figures out the x (user input) popular stories based on the number of
    RATs
91.      * that participated in them.
92.      *
93.      * @param x how many stories to return
94.      * @return a string representation of the x most popular stories
95.      */
96.     protected String mostPopularStories(int x) {
97.         Iterator<String> iterator = (g.stories.keySet()).iterator(); //iterates
    over stories keys
98.         String s = "";
99.
100.            int count = 0; //counts how many stories have been added. Makes it
    possible to include
101.            //values that are tied for the highest participation that would
    exceed
102.            //the user input
103.
104.            while(iterator.hasNext()) {
105.                String key = iterator.next(); //gets next key
106.                if (g.stories.get(key) == highestStories) { //if the story count
    was the highest, adds it!
107.                    count++;
108.                    s += ("\nStory: "  + key
```

```
109.                        + "\t\tTimes Tweeted About: "  + g.stories.get(key));
110.                    mostPopStories.put(key, g.stories.get(key));
111.                }
112.            }
113.
114.            highestStories--; //subtracts by one to find the next highest stories
115.
116.            if (count < x) { //if we haven't found x RATs, calls
    mostPopularStories recursively
117.                s += mostPopularStories(x - count);
118.            }
119.
120.            return s;
121.        }
122.
123.        /**
124.         * Figures out the x (user input) popular stories based on the number of
    RATs
125.         * that participated in them.
126.         *
127.         * @param x how many stories to return
128.         * @return a string representation of the x most popular stories
129.         */
130.        protected String leastPopularStories(int x) {
131.            Iterator<String> iterator = (g.stories.keySet()).iterator();
    //iterates over stories keys
132.            String s = "\n   Stories with " + lowestRATs + " participating:\n
    \t";
133.
134.            int count = 0; //counts how many stories have been added. Makes it
    possible to include
135.            //values that are tied for the highest participation that would
    exceed
136.            //the user input
137.
138.            while(iterator.hasNext()) {
139.                String key = iterator.next(); //gets next key
140.                if (g.stories.get(key) == lowestStories) { //if the story count
    was the lowest, adds it!
141.                    count++;
142.                    s += (key + ", ");
143.                    leastPopStories.put(key, g.stories.get(key));
144.                }
145.            }
146.
147.            lowestStories++; //subtracts by one to find the next lowest stories
148.
149.            if (count < x) { //if we haven't found x RATs, calls
    mostPopularStories recursively
150.                s += leastPopularStories(x - count);
```

```
151.            }
152.
153.            s = s.substring(0, (s.length() - 2));
154.
155.            return s;
156.        }
157.
158.        /**
159.         * Gets the number of stories that each RAT participated in and
    determines
160.         * what the highest and lowest number of RATs per story is.
161.         */
162.        protected void storiesPerRAT() { //could just use story_count...
163.            // Collection Iterator
164.            Iterator<String> iterator = g.accounts.keySet().iterator();
    //iterates over accounts keys
165.
166.            highestRATs = 0;
167.            lowestRATs = 5000; // there are fewer than 5000 RATs, so assumes this
    is high enough
168.
169.            while(iterator.hasNext()) {
170.                String key = iterator.next(); //gets next ket
171.                RAT current = g.accounts.get(key); //gets next RAT
172.
173.                int numStories = g.accounts.get(key).stories.size(); //gets # of
    stories the RAT
174.                //participated in
175.                storiesPerRATValues.add(numStories);//adds the number to a
    LinkedList of values for
176.                //later use
177.                if (numStories >= highestRATs) { //if the story count is higher,
    updates highestRATs
178.                    highestRATs = numStories;
179.                }
180.
181.                if (numStories <= lowestRATs) { //if the story count is lower,
    updates lowestRATs
182.                    lowestRATs = numStories;
183.                }
184.            }
185.        }
186.
187.        /**
188.         * Figures out the x (user input) most active RATs based on the number of
    stories
189.         * each RAT participated in
190.         *
191.         * @param x how many RATs to return
192.         * @return a string representation of the x most active RATs
```

```
193.         */
194.        protected String mostActiveRATs(int x) {
195.            Iterator<String> iterator = (g.accounts.keySet()).iterator();
    //iterates over accounts keys
196.            String s = "";
197.
198.            int count = 0;  //counts how many RATs have been added. Makes it
    possible to include
199.            //values that are tied for the highest participation that would
    exceed
200.            //the user input
201.            while(iterator.hasNext()) {
202.                String key = iterator.next(); //gets next key
203.                RAT current = g.accounts.get(key); //gets next RAT
204.                int numStories = g.accounts.get(key).stories.size(); //how many
    stories the RAT
205.                //participated in
206.                if (numStories == highestRATs) { //sees if RAT participated in
    the highest # of stories
207.                    count++;
208.                    s += ("\nRat: "  + key
209.                        + "\t\tNumber of Stories Tweeted About: "  + numStories);
210.                    mostActiveRATs.put(key, current);
211.                }
212.            }
213.
214.            highestRATs--; //decrements by one to check next highest
    participation level
215.
216.            if (count < x) { //if we haven't found x RATs, calls mostActiveRATs
    recursively
217.                s += mostActiveRATs(x - count);
218.            }
219.
220.            return s;
221.        }
222.
223.        /**
224.         * Figures out the x (user input) most active RATs based on the number of
    stories
225.         * each RAT participated in
226.         *
227.         * @param x how many RATs to return
228.         * @return a string representation of the x most active RATs
229.         */
230.        protected String leastActiveRATs(int x) {
231.            Iterator<String> iterator = (g.accounts.keySet()).iterator();
    //iterates over accounts keys
232.            String s = "\n   Tweeted about " + lowestRATs + " stories:\n   \t";
233.
```

```
234.            int count = 0;  //counts how many RATs have been added. Makes it
    possible to include
235.            //values that are tied for the highest participation that would
    exceed
236.            //the user input
237.            while(iterator.hasNext()) {
238.                String key = iterator.next(); //gets next key
239.                RAT current = g.accounts.get(key); //gets next RAT
240.                int numStories = g.accounts.get(key).stories.size(); //how many
    stories the RAT
241.                //participated in
242.                if (numStories == lowestRATs) { //sees if RAT participated in the
    highest # of stories
243.                    count++;
244.                    s += (key + ", ");
245.                }
246.            }
247.
248.            lowestRATs++; //decrements by one to check next highest participation
    level
249.
250.            if (count < x) { //if we haven't found x RATs, calls mostActiveRATs
    recursively
251.                s += mostActiveRATs(x - count);
252.            }
253.
254.            s = s.substring(0, (s.length() - 2));
255.
256.            return s;
257.        }
258.
259.        /**
260.         * Figures out the average and median stories participated in per RAT AND
261.         * the average and median RATs per story. Prints both values. Assumes
    that
262.         * the lists will only be the ones created by this Investigate class.
    Throws
263.         * an error if an invalid list is given, but it still won't work on any
    list
264.         * except the specified two.
265.         *
266.         * @param list the list to find the values from
267.         */
268.        protected void normalDistribution(LinkedList<Integer> list) {
269.            int sum = 0;
270.
271.            for (int value : list) {
272.                sum += value;
273.            }
274.
```

```
275.          if (list.equals(storiesPerRATValues)) {
276.
277.              if (storiesPerRATValues.size()%2 == 0) {
278.                  medStoriesParticipated = list.get(list.size() / 2);
279.              } else {
280.                  medStoriesParticipated = list.get(list.size() / 2 + 1);
281.              }
282.
283.              avgStoriesParticipated = sum / list.size();
284.
285.              System.out.println("\nThe highest number of RATs that
    participated in a story is: " +
286.                  highestStories);
287.              System.out.println("The lowest number of RATs that participated
    in a story is: " +
288.                  lowestStories);
289.              System.out.println("The median number of RATs that participated
    in each story is: " +
290.                  medStoriesParticipated);
291.              System.out.println("The average number of of RATs that
    participated in each story is: " +
292.                  avgStoriesParticipated);
293.          } else if (list.equals(RATsPerStoryValues)) {
294.
295.              if (RATsPerStoryValues.size()%2 == 0) {
296.                  medRATsPerStory = list.get(list.size() / 2);
297.              } else {
298.                  medRATsPerStory = list.get(list.size() / 2 + 1);
299.              }
300.
301.              avgRATsPerStory = sum / list.size();
302.
303.              System.out.println("\nThe highest number of stories tweeted about
    by a RAT is: " +
304.                  highestRATs);
305.              System.out.println("The lowest number of stories tweeted about by
    a RAT is: " +
306.                  lowestRATs);
307.              System.out.println("The median number of stories tweeted about by
    the RATs is: " +
308.                  medRATsPerStory);
309.              System.out.println("The average number of stories tweeted about
    by the RATs is: " +
310.                  avgRATsPerStory);
311.          } else {
312.              System.out.println("Please input a valid list.");
313.          }
314.      }
315.
316.      /**
```

```
317.         * Uses breadth first search to find the diameter of the graph
318.         */
319.        public int getDiameter() {
320.            Iterator<String> iterator = g.accounts.keySet().iterator();
    //iterates over accounts keys
321.            diameter = 0; //initial diameter, expected to grow
322.
323.            boolean checked = false;
324.            int levels = 0;
325.
326.            while(iterator.hasNext()) {
327.                String key = iterator.next();
328.                RAT current = g.accounts.get(key);
329.
330.                LinkedList<String> currentBF =
    g.graph.BFtraversal(current.username); //BFS on current RAT
331.                String first = currentBF.remove();
332.                levels = 0;
333.
334.                for (String element : currentBF) {
335.                    if ((g.accounts.containsKey(element)) && (checked == false))
    {
336.                        checked = true;
337.                        levels++;
338.                    } if (!g.accounts.containsKey(element) && (checked == true))
    {
339.                        checked = false;
340.                        levels++;
341.                    }
342.                }
343.
344.                if (levels >= diameter) {
345.                    diameter = levels; //diameter grows as soon as the BFS
    produces a larger value
346.                }
347.            }
348.
349.            return levels;
350.        }
351.
352.        /**
353.         * Uses breadth first search to find the radius of the graph
354.         */
355.        public int getRadius() {
356.            Iterator<String> iterator = g.accounts.keySet().iterator();
    //iterates over accounts keys
357.            radius = diameter; //initial diameter, expected to shrink
358.
359.            boolean checked = false;
360.            int levels = 0;
```

```
361.
362.            while(iterator.hasNext()) {
363.                String key = iterator.next();
364.                RAT current = g.accounts.get(key);
365.
366.                LinkedList<String> currentBF =
    g.graph.BFtraversal(current.username); //BFS on current RAT
367.                String first = currentBF.remove();
368.                levels = 0;
369.
370.                for (String element : currentBF) {
371.                    if ((g.accounts.containsKey(element)) && (checked == false))
    {
372.                        checked = true;
373.                        levels++;
374.                    } if (!g.accounts.containsKey(element) && (checked == true))
    {
375.                        checked = false;
376.                        levels++;
377.                    }
378.                }
379.
380.                if (levels == radius) {
381.                    centerNodes.add(current.username); //adds the user to the
    list if they have an eccentricity that
382.                                                       //matches the current
    eccentricity
383.                }
384.                if (levels < radius) { //if the user has an eccentricity smaller
    than the current eccentricity...
385.                    radius = levels; //radius shrinks as soon as the BFS produces
    a smaller value
386.                    centerNodes.clear(); //clears the previous centerNodes list
387.                    centerNodes.add(current.username); //adds the current user to
    the now-empty centerNodes list
388.                }
389.            }
390.
391.            return radius;
392.        }
393.
394.        /**
395.         * Uses depth first search to determine whether the graph is connected or
    not.
396.         * Uses the first RAT just for ease of use, then compares all other RATs
    to the
397.         * list of RATs provided by the DFS to determine whether the graph is
    connected or not.
398.         *
399.         * @return a boolean indicating the graph's connectivity
```

```
400.         */
401.        public boolean isConnected() {
402.            Iterator<String> iterator1 = g.accounts.keySet().iterator();
    //iterates over the keys
403.            Iterator<String> iterator2 = g.stories.keySet().iterator();
    //iterates over the keys
404.            String key = iterator1.next(); //gets the first RAT's key
405.            RAT current = g.accounts.get(key); //gets the first RAT
406.
407.            LinkedList DFresults = g.graph.DFtraversal(current.username);
    //performs DFS on the RAT
408.
409.            while(iterator1.hasNext()) {
410.                key = iterator1.next();
411.                if (!DFresults.contains(key)) {
412.                    return false; //returns false as soon as a RAT in the graph
    is not found in the
413.                    //DFS linked list
414.                }
415.            }
416.
417.            while(iterator2.hasNext()) {
418.                key = iterator2.next();
419.                if (!DFresults.contains(key)) {
420.                    return false; //returns false as soon as a RAT in the graph
    is not found in the
421.                    //DFS linked list
422.                }
423.            }
424.
425.            return true; //returns true if all RATs were found in the DFS linked
    list
426.        }
427.
428.        /**
429.         * Determines how many RATs participated in both stories passed in as
430.         * parameters. Used to figure out if popular stories were often tweeted
431.         * about by the same RATs or if the groups differed.
432.         *
433.         * @param story1 the first story
434.         * @param story2 the second story
435.         *
436.         * @return a list of RATs that tweeted about both stories
437.         */
438.        public LinkedList<String> compareRATs(String story1, String story2) {
439.            Iterator<String> iterator = g.accounts.keySet().iterator();
    //iterates over the keys
440.
441.            LinkedList<String> overlaps = new LinkedList<String>();
442.
```

```
443.            while(iterator.hasNext()) {
444.                String key = iterator.next(); //gets the first RAT's key
445.                RAT current = g.accounts.get(key); //gets the first RAT
446.                if ((current.stories).contains(story1) &&
    (current.stories).contains(story2)) {
447.                    overlaps.add(current.username);
448.                }
449.            }
450.
451.            return overlaps;
452.        }
453.
454.        /**
455.         * Determines how many RATs "overlapped" in the most popular stories.
456.         *
457.         * @return a string representation of the overlapping rats
458.         */
459.        public String overlappingRATs(Hashtable<String,Integer> hash) {
460.            Iterator<String> iterator = hash.keySet().iterator(); //iterates over
    the keys
461.            String[] storyIDs = new String[hash.size()];
462.            int storyIDSize = 0;
463.
464.            while(iterator.hasNext()) {
465.                String key = iterator.next(); //gets the first RAT's key
466.                storyIDs[storyIDSize++] = key;
467.            }
468.
469.            String output = "";
470.
471.            for (String s1 : storyIDs) {
472.                LinkedList<String> currentOverlaps = new LinkedList<String>();
473.                output += "\n\nStory " + s1 + ":";
474.                for (String s2: storyIDs) {
475.                    if (!s1.equals(s2)) {
476.                        currentOverlaps = compareRATs(s1, s2);
477.                        output += "\n    Story " + s2 + " (" +
    currentOverlaps.size() + " overlaps):\t" + currentOverlaps;
478.                    }
479.                }
480.            }
481.
482.            return output;
483.        }

484.    public String getStoryData() { //Written by Efua Akonor
485.            //Saves csv on all stories and their participation/popularity
486.
487.            //create csv and store data in it
488.            try {
```

```
489.            FileWriter csvWriter = new FileWriter("StoryData.csv");
490.
491.
492.            Iterator<String> iterator = (g.stories.keySet()).iterator();
    //iterates over stories keys
493.            csvWriter.append("story_ID, tweet_count");
494.
495.            int count = 0; //counts how many stories have been added. Makes it
    possible to include
496.                           //values that are tied for the highest participation
    that would exceed
497.                           //the user input
498.
499.            while(iterator.hasNext()) {
500.                String key = iterator.next(); //gets next key
501.                csvWriter.append("\n"+ key + ", "  + g.stories.get(key));
502.
503.            }
504.
505.            csvWriter.flush();
506.            csvWriter.close();
507.
508.            return "exported.";
509.
510.            } catch(IOException e) {
511.                e.printStackTrace();
512.                return "error.";
513.            }
514.        }
515.
516.        public String getRatData() { //Written by Efua Akonor
517.            //Saves csv on all RATS and their participation/popularity
518.            try {
519.            FileWriter csvWriter = new FileWriter("RATData.csv");
520.            Iterator<String> iterator = (g.accounts.keySet()).iterator();
    //iterates over accounts keys
521.            csvWriter.append("rat_ID, story_count");
522.
523.            while(iterator.hasNext()) {
524.                String key = iterator.next(); //gets next key
525.                RAT current = g.accounts.get(key); //gets next RAT
526.                int numStories = g.accounts.get(key).stories.size(); //how many
    stories the RAT
527.
    //participated in
528.                csvWriter.append("\n"  + key + ", "  + numStories);
529.            }
530.            csvWriter.flush();
531.            csvWriter.close();
532.
```

```
533.            return "exported.";
534.        } catch(IOException e) {
535.                e.printStackTrace();
536.                return "error.";
537.            }
538.        }

539.
540.        public static void main(String[] args) {
541.            Investigate i = new
    Investigate("All_Russian-Accounts-in-TT-stories.csv.tsv", 10);
542.
543.            i.RATsPerStory();
544.            i.storiesPerRAT();
545.
546.            //prints the total number of RATs and total number of stories
547.            System.out.println("Total number of RATs: " + i.g.usernamesSize);
548.            System.out.println("Total number of stories: " + i.g.stories.size());
549.
550.            //calculates the median and average RATs-per-story and
    stories-per-RAT values
551.            i.normalDistribution(i.storiesPerRATValues);
552.            i.normalDistribution(i.RATsPerStoryValues);
553.
554.            //figures out (and prints) the most and least active RATs, determines
555.            //what the most popular stories were about
556.            System.out.println("\nThe " + i.numPops + " most popular stories
    were:"
557.                + i.mostPopularStories(i.numPops));
558.            System.out.println("\nThe most popular stories were about: ");
559.            TwitterTrails t = new TwitterTrails(i.mostPopStories);
560.            System.out.println("\nThe " + i.numPops + " least popular stories
    were:"
561.                + i.leastPopularStories(i.numPops));
562.
563.            //figures out (and prints) the most and least popular stories
564.            System.out.println("\nThe " + i.numPops + " most active RATs were:"
565.                + i.mostActiveRATs(i.numPops));
566.            System.out.println("\nThe " + i.numPops + " least active RATs were:"
567.                + i.leastActiveRATs(i.numPops));
568.
569.            i.getDiameter();
570.            i.getRadius();
571.
572.            System.out.println("\nThe diameter of the graph is: " + i.diameter);
573.            System.out.println("\nThe radius of the graph is: " + i.radius);
574.            System.out.println("\nThe center nodes of the graph are: " +
    i.centerNodes);
575.
576.            System.out.println("\nThe graph is connected: " + i.isConnected());
```

```
577.
578.            System.out.println("How the 10 most popular stories overlapped with
    eachother:\n" + i.overlappingRATs(i.mostPopStories));
579.
580.            i.g.exportAccounts("accounts.csv");
581.            i.g.exportStories("stories.csv");
582.        }
583.    }
```

Collaboration

Introduction

Context on project objective provided by Efua Akonor. Context on historical background, the RATs, and RAT activity provided by Kathryn Swint.

Conclusions

Initial RAT and story numbers collected by Efua Akonor in CSV files (included in final project submission folder). Data on individual RATs, results from the Investigate file, information about individual RATs found online, and analysis of the findings provided by Kathryn Swint.

Code & Methods

AdjListGraph.java, RAT.java, RATgraph.java, Webpage.java, and TwitterTrails.java developed by Kathryn Swint. Webpage.java and TwitterTrails.java came from PS05, and were written in collaboration with Anushe Sheikh. Methods section provided by Kathryn Swint. Investigate.java developed by Efua Akonor and Kathryn Swint.