Beyond Keywords: Tracking the evolution of conversational clusters in social media

James P. Houghton
Michael Siegel
Stuart Madnick
Nobuaki Tounaka
Kazutaka Nakamura
Takaaki Sugiyama
Daisuke Nakagawa
Buyanjargal Shirnen

Working Paper CISL# 2016-13

October 2017

Cybersecurity Interdisciplinary Systems Laboratory (CISL)
Sloan School of Management, Room E62-422
Massachusetts Institute of Technology
Cambridge, MA 02142

Beyond Keywords: Tracking the evolution of conversational clusters in social media

James P. Houghton[1], Michael Siegel[1], Stuart Madnick[1], Nobuaki Tounaka[2], Kazutaka Nakamura[2], Takaaki Sugiyama[2], Daisuke Nakagawa[2], and Buyanjargal Shirnen[2]

[1]Massachusetts Institute of Technology, Cambridge, MA, USA
[2]Universal Shell Programming Laboratory Ltd., Tokyo, Japan

October 9, 2017

Abstract

The potential of social media to give insight into the dynamic evolution of public conversations, and into their reactive and constitutive role in political activities, has to date been underdeveloped. While topic modeling can give static insight into the structure of a conversation, and keyword volume tracking can show how engagement with a specific idea varies over time, there is need for a method of analysis able to understand how conversations about societal values evolve and react to events in the world, incorporating new ideas and relating them to existing themes. In this paper, we propose a method for analyzing social media messages that formalizes the structure of public conversations, and allows the sociologist to study the evolution of public discourse in a rigorous, replicable, and data-driven fashion. This approach may be useful to those studying the social construction of meaning, the origins of factionalism and internecine conflict, or boundary-setting and group-identification exercises; and has potential implications for those working to promote understanding and intergroup reconciliation.

1 Motivation

Social media suggests a tantalizing prospect for researchers seeking to understand how newsworthy events influence public and political conversations. Messages on platforms such as Twitter and Facebook represent a high volume sample of the national conversation in near real time, and with the right handling[1] can give insight into events such as elections, as demonstrated by [Huberty, 2013, Tumasjan et al., 2010]; or processes such as social movements, as demonstrated by [Agarwal et al., 2014, DiGrazia, 2015]. Standard methods of social media analysis use keyword tracking and sentiment analysis to attempt to understand the range of perceptions surrounding these events. While helpful for basic situational awareness, these methods do not help us understand how a set of narratives compete to interpret and frame an issue for action.

For example, if we are interested in understanding the national conversation in reaction to the shooting at Emanuel AME church in Charleston, South Carolina on June 17, 2015, we could plot a timeseries of the volume of tweets containing the hashtag `#charleston`, as seen in Figure 1. This tells us that engagement with the topic spiked on the day of the news, and then persisted for several days before returning to baseline. It does not tell us that the initial spike may have been retellings of the details of the event

[1]The primary issues involve controlling for demographics. For examples of methods for handling demographic concerns with social media data, see [Bail, 2015, McCormick et al., 2015].

itself, with the tail comprised of interpretations and framings of the event in the political discourse, the ways the event is interpreted by various groups, or how these groups connect the event to their preexisting ideas about gun violence. Additionally, it does not let us track continuing engagement with the follow-on ideas stimulated by the original discussion.

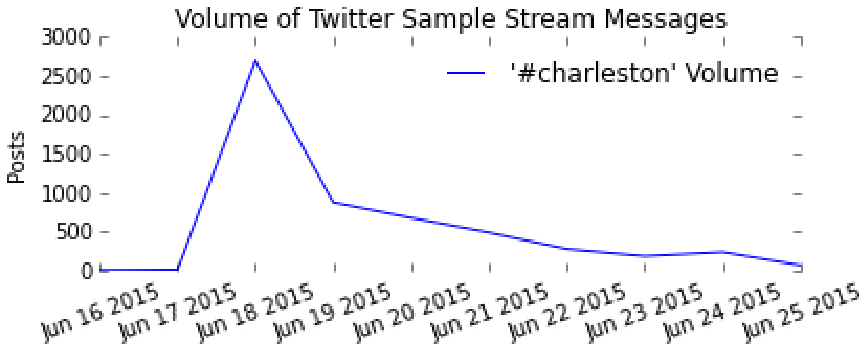

Figure 1: Standard methods of social media analysis include keyword volume tracking[3](shown here), sentiment analysis, and supervised categorization

Alternate methods include assessing the relative ‘positive’ or ‘negative’ sentiment present in these tweets, or using a supervised learning categorizer to group messages according to preconceived ideas about their contents, as demonstrated by [Becker et al., 2011, Ritter et al., 2010, Zubiaga et al., 2011]. Such methods can give aggregated insight into the sentiment expressed, but fail to uncover from the data the ways the events are being framed within existing conversations.

[3]Data presented here (and in the remainder of this paper) is from a 1% random sample of twitter messages. The number of messages listed in the y axis of this figure (and other count based figures) represents volume within the sample. For an estimate of the overall volume, multiply by 100.

Because they look at counts of individual messages and not at the relationships between ideas, techniques which focus on volume, sentiment, or category are unable to uncover coherent patterns of thought that signify a society's interpretation of the events' deeper meanings. To interpret events, individuals must make connections between an event and historical parallels or concepts in the public discourse. It is thus the expressed connections between ideas, not merely the ideas themselves, which must be tracked, categorized, and interpreted as samples from an underlying semantic structure. Our approach inverts the investigation of structures of connected persons observed in *social* networks, as demonstrated by [Zachary, 1977] or [Morales et al., 2015], and instead investigates structures of connected ideas within *semantic* networks.[4]

While the individual-level semantic networks which constitute personal interpretation of events are fundamentally unobservable, messages sent by individuals encode samples of these connections in an observable format. These individual-level samples can be aggregated to form a societal-level network representing the superposition of active semantic network connections of the society's members. For the purpose of this paper, it is sufficient to note that if structure is discernible in this aggregate then structure is implied within the members, allowing that no one individual need represent more than a subset of the aggregate structure.[5]

[4]For a theoretical discussion of semantic networks as representations of human knowledge, see [Woods, 1975], [Mayer, 1995] and [Schilling, 2005], for a physiological description see [Tulving, 1972] and [Collins and Quillian, 1969].

[5]The fact that macroscopic structures do exist in the aggregate representation is itself

One way to aggregate these connections is to look at a network of word co-occurrences in social media messages, as has been demonstrated by [Cogan and Andrews, 2012, Smith and Rainie, 2014]. This technique represents references to an event, to analogous historical events, and to other public discourse concepts as nodes in a semantic network. Each message containing two concepts contributes to the weight of an edge between these nodes. The structure that results gives a macro-level aggregated sample of the micro-level structures of meaning within individuals' own minds. For example, Figure 2 shows the network of hashtags formed around the focal concept of the Charleston shooting, on June 18, 2015. Each hashtag in the conversation is represented as a point or 'node' in this diagram, and each message containing a pair of hashtags contributes to the strength of the link or 'edge' connecting those two nodes.[6]

Within this example appear two distinct clusters of connected ideas. In this case, the upper cluster represents a description of the shooting itself and the human elements of the tragedy, and the lower cluster focusses on the larger national-scale political conflicts to which the event relates. Within each cluster, ideas relate to one another, and connections give context to and comment upon one another. We might consider this the essence of

surprising. One could imagine that such an aggregate would resemble a random network, and the fact that this is not the case suggests an underlying sociological process for the social construction of meaning. A full description of how these processes may operate is forthcoming by the first author.

[6]Here we use a cutoff threshold to convert a frequency-weighted set of connections into an unweighted network diagram.

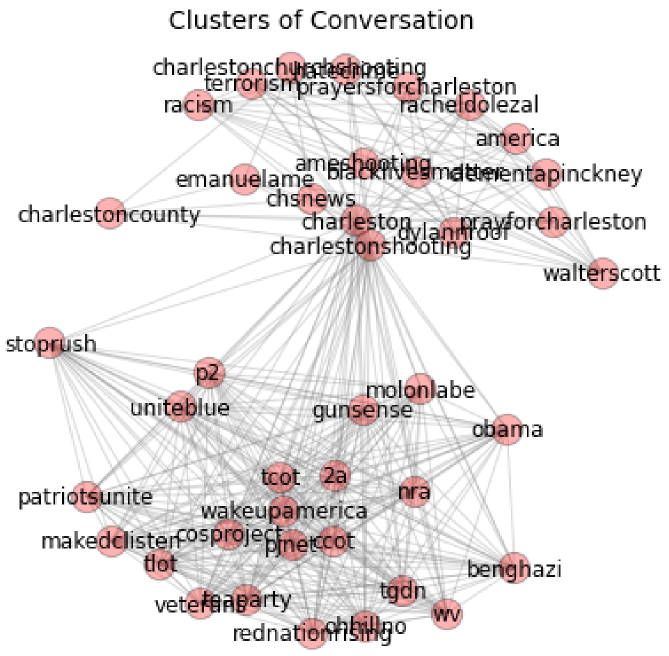

Figure 2: Coherent structures in the aggregate hashtag co-occurrence network can serve as proxies for 'conversations' in the larger societal discourse, and give insight into the structures of meaning within the minds of individual members of that society

what is meant by a national conversation around a topic - a set of mutually acknowledged statements of meaning. In contrast, across clusters we see fewer connections between ideas, suggesting that the statements made within one cluster do not inform or relate to those from the other. By examining the structure of the semantic network, we begin to see that instead of one national conversation reacting to the event, there are (at least) two conversations going on, each strongly connected to itself, but only weakly relating to the other.

1.1 The contributions of this paper

We may hypothesize that how these conversations evolve and interact will influence the social and political response to the event they describe. In this paper we build on the qualitative example above, and present a formal method for identifying conversational clusters, describing their structures, and tracking their development over time. We present various methods for visualizing conversational structure to give insight into how certain elements form the central themes of a conversation while other elements circulate on the margins. We show how overlapping conversational clusters may be identified, and explore their relevance to discussions of factionalism and consensus building. We then demonstrate a method for tracking the evolution of these conversational clusters over time, in which clusters identified on one day are compared with clusters emerging on subsequent days. This allows us to identify how new concepts are being included into the discussion, and quantitatively track engagement in full conversations as distinct from mere references to keywords.

Formally representing elements of a conversation as being frequent vs. rare, or central vs. marginal, and tracking how these metrics develop over time, allows the sociologist to study the way new ideas obtain relevance. For example, a central and well-connected topic may become prominent through a process of amplification, or a related popular topic may become relevant through a process of linking. These phenomena, which happen societally at the level of individual terms, can then be studied as drivers of macro-level phenomena such as factionalism, polarization, and realignment. By tying the changes in conversation to changes in underlying social structures, the narrative identity-forming and boundary-setting activities of groups may be studied.

In the appendices to this paper we provide example scripts that save those wishing to use these techniques from the burden of reimplementing these algorithms. Due to the computation-intensive nature of this analysis, we chose to implement the data manipulation algorithms in both Python and Unicage shell scripts,[7] for prototyping and speed of execution, respectively. Descriptions of these scripts can be found in the appendices, along with performance comparisons between the two languages.

2 Identifying Conversation Clusters

The network in Figure 2 represents connections individuals have made between hashtags.[8] Each of the hashtags present in the dataset forms a node in this network, and the relative strength of edges depends upon the number of times the pair occur together in a tweet, their 'co-occurrence', using the method of [Marres and Gerlitz, 2014].[9] To formally identify the clus-

[7]For a description of Unicage development tools, see [Tounaka, 2013]

[8]The attributes of these connections are not considered in this analysis, and may thus be positive or negative, binding or exclusive.

[9]It is of course possible to conduct the analysis using the full set of words present in a tweet, omitting stop-words, or focussing purely upon easily identifiable concepts. From an analytical perspective, this has the effect of expanding the scale of the clusters, as a broader range of concepts are included. Within this paper, we limit the analysis to hashtags purely for the purposes of simplifying the presentation, and keeping visualizations to an

ters visually present in the diagram, we apply k-clique community detection algorithms, implemented in the COS Parallel library developed by [Gregori et al., 2013] and demonstrated in the appendices.

Clique percolation methods work to identify communities of well-connected nodes within a network in a way that allows for the possibility of overlapping or nested communities. The methods allow the analyst to create metrics that identify communities based upon local network characteristics and are thus invariant to network size or connectivity outside of the local community. As the measured extent of a conversation should be determined by the content and connectivity of that conversation, and not by the structure of unrelated discussion, this method of community detection is for our purposes superior to those based upon graph cutting or intersubjective distance measures.[10]

The K-clique percolation method suggests that communities are composed of an interlocking set of cliques of size `k`, that is `k` nodes which are all mutually connected to each other; and that the overlap of each clique with a neighboring clique is `(k-1)`. This ensures that each member of a community is connected to at least `(k-1)` other mutually-connected nodes within the community, and that any sub-communities are connected by a bridge of at

appropriate size. We are grateful to an anonymous reviewer for highlighting that a full text analysis not only allows for a larger sample of messages to be observed within the conversation, but is essential for researchers studying how concepts rise to prominence, as the initial process of concept formation or linkage is likely to occur before the concepts are institutionalized with a hashtag. A comparison of the full-text and hashtag-only analysis is present in Appendix F.

[10]For a comparison of community detection algorithms and their features, see [Porter et al., 2009].

least width `k`.

The metric `k` determines how strict the conditions are for membership in a cluster, and thus drives cluster extent and boundaries. For example, high values of `k` would impose strict requirements for interconnectedness between elements of an identified conversation, leading to a smaller, more coherent identified conversation, as seen in Figure 3. Each member of this cluster is connected to at least 4 other members who are all connected to one another.

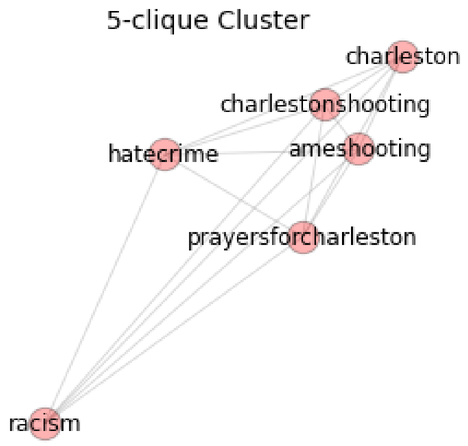

Figure 3: Clusters with higher `k` value are smaller and more tightly connected, representing a more coherent or focused conversation

On the other hand, smaller values of `k` are less stringent about the requirements of connectivity they put on the elements in the cluster, leading to a larger, more loosely coupled group as seen in Figure 4, whose members must be connected to at least two other mutually connected members of the community.

When we apply these methods to word co-occurrence networks, the result

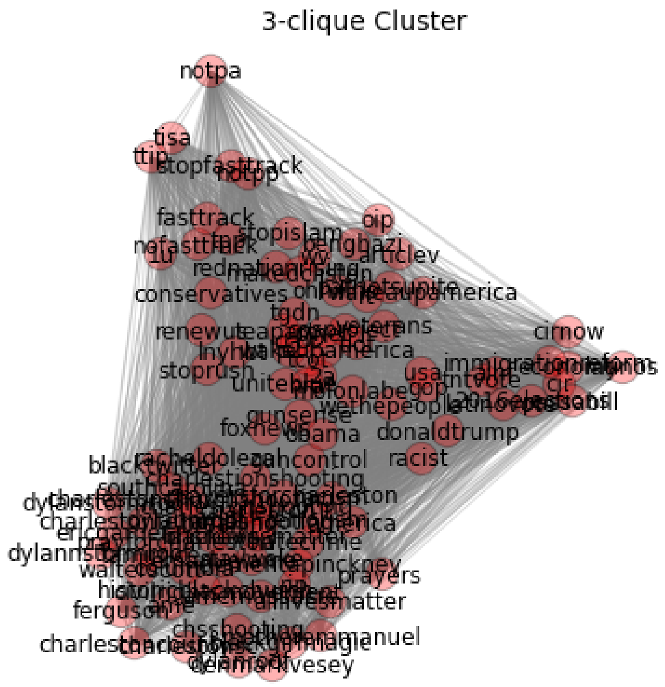

Figure 4: Clusters with lower `k` value are larger and less less tightly connected, representing more diffuse conversation. They may have smaller clusters of conversation within them

is a collection of sets of words which all form a conversational cluster. As the extent and connectivity of the identified cluster is dependent upon the value of `k` used to create it, by varying `k` we can identify closely connected subsets that represent the central structures of the conversation, and occasionally multiple subsets representing internal factions within the discussion.

As the algorithms depend only on the presence, not the strength of connections within the conversation, information of the volume of messages making a semantic connection can be encoded as a threshold weight `w` for inclusion into the analyzed co-occurrence network. By varying `w` we can identify the extent to which a connection is recognized by members of the population. To demonstrate the utility of these parameters we explore their extrema. Clusters that form with high values of `k` and low values of `w` may be considered central to the conversation, but only by a minority of the population. Conversely, concepts contained only within clusters with high values of `w` and low values of `k` are universally agreed to be marginally related to the focal discussion.

3 Representing Conversational Clusters as Nested Sets

Tight conversational clusters (high `k`) must necessarily be contained within larger clusters with less stringent connection requirements (lower `k`). Performing clustering along a range of `k` values allows us to place a specific conversation in context of the larger discourse. It becomes helpful to represent these clusters as nested sets as seen in Figure 5, ignoring the node and edge construction of the network diagram in order to display the nested and interlocking relationships the conversations have with one another. In this diagram, members of a 5-clique cluster are arranged (in no particular order) within the darker blue box. These form a subset of a larger 4-clique cluster which also includes a number of other concepts.

With this representation we see elements of the conversation that are more well-connected and thus 'central' to the conversation, along with those

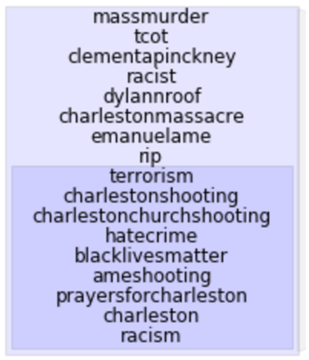

Figure 5: Converting networks to nested sets based upon k-clique clustering simplifies presentation and analysis of various levels of conversation

relating to one another less tightly. Existing methods which look at volume may suggest that ideas are central to a conversation if they are well represented in the sample. These methods lack the ability to analytically discern that different topics are located within the core of different conversations. Within the nested structure presented here, central topics are defined not by their volume, but by the multiplexity of their connection with other concepts which also form the core of the discussion, elements to which they relate and whose meaning they are seen as essential to. Within the larger conversation, peripheral terms are less mutually interdependent, and less essential for understanding.

Specifically, this method allows the qualitative observation of conversational clusters that we drew from Figure 2 to be formalized into an explicit and analytically tractable structure. The upper cluster of our example is highlighted here, and we show that this cluster is actually composed of a more tightly connected set of central ideas referencing the event itself and the immediate response it elicits, embedded within a broader set of ideas additionally referencing the perpetrator and victims.[11] In noting that the abstractions of terrorism and racism are part of the center of this part of the conversation, with the specific details closer to the margins, the sociologist may wonder if this is because the event is seen primarily as an instance of a more important phenomena than as an interesting occurrence in its own right.

4 Tracking Conversations Chronologically

In order to track how elements of conversation weave into and out of the general discourse over time, we need to be able to interpret how conversational clusters identified at one point in time relate to those in subsequent intervals. We can do this in one of two ways.

The first method is to track the volume of co-occurrences identified in the various conversational clusters identified on the first day of the analysis, as it changes over subsequent days. This indicates how well the connections made on the first day maintain their relevance in the larger conversation. Figure 6 shows how the connections made in conversational clusters on June 18th fall in volume over the 10 days subsequent to the initial event, paralleling the

[11]This also reveals that ideas which at first seem related, such as ‘racism’ and ‘racist’ might hold different valences. This diagram would suggest that systemic racism attributed to the event is more closely connected with other details of the conversation than is its specific manifestation in the racist perpetrator. The robustness of these conclusions would need to be explored by further varying the thresholds used to construct these clusters.

decay in pure keyword volume seen in Figure 1.

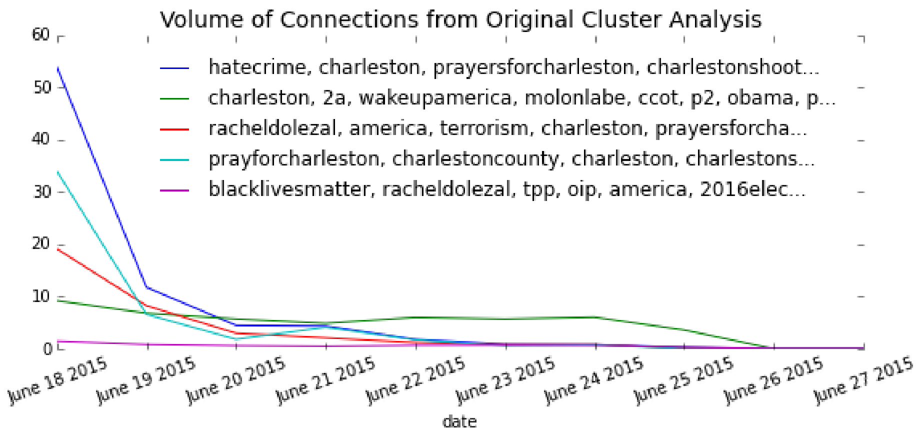

Figure 6: Tracking the volume of connections made in a single day's clusters (e.g. co-occurrences) reveals how the specific analogies made immediately after the event maintain their relevance

The second method for tracking conversation volume over time takes into account the changes that happen within the conversation itself. The fundamental assumption in this analysis is that while the words and connections present in a conversation change, they do so incrementally in such a way as to allow for matching conversations during one time period with those in the time period immediately subsequent.[12]

[Palla et al., 2007] discuss how communities of individuals develop over time and change. We can use the same techniques to track the continuity of conversational clusters. The most basic way to do this is to count the overlap

[12]For an intuitive analogy, consider that a fraction of your favorite baseball team may be replaced in any given year, but the 'team' persists. It would be easy to identify these persisting teams by matching rosters from one season with those from the preceding and subsequent seasons, even if particular players transfer to other teams in the league.

of elements of conversational clusters at time 1 and time 2, as a fraction of the total number of elements between the two, and use this fraction as the likelihood that each cluster at time 2 is an extension or contraction of the time 1 cluster in question. From this we can construct a transition matrix relating conversational clusters at time 1 with clusters at time 2.[13]

For instance, if we wish to understand the likelihood that the outermost cluster illustrated in Figure 5 transitions to a subsequent cluster on June 19th, as visualized in Figure 7, we can count the number of words present in both days (7: 'terrorism', 'dylannroof', 'charlestonshooting', etc...) and divide by the total unique number of words in the clusters on both days combined (19: adding 'whitesupremacy', 'massmurder', etc...), giving a value of 0.37.[14]

This value forms the entry in our transition matrix with row 20150618-cl1, and column 20150619-cl1, excerpted in Table 1. Other clusters which are candidates for succeeding our focal cluster form other entries in the same row. Other clusters from the first day and their candidates for transition form the additional rows.

[13]In this analysis, we form clusters using all messages sent on a given day, and look at transitions from day to day. This is appropriate as the underlying sociological process we are interested in occurs at this timescale. For slower-changing conversations, it may be useful to construct clusters by aggregating messages to the weekly or monthly level, and computing transitions between these times

[14]To improve our estimates, we can take advantage of the fact that clusters that correspond from time 1 to time 2 will participate in a larger cluster that emerges if we perform our clustering algorithm on the union of all edges from the networks at time 1 and time 2. Omitting entries in the transition matrix which do not nest within this joint cluster reduces the number of possible pairings across days, yielding a sparser and more specific intra-day transition matrix.

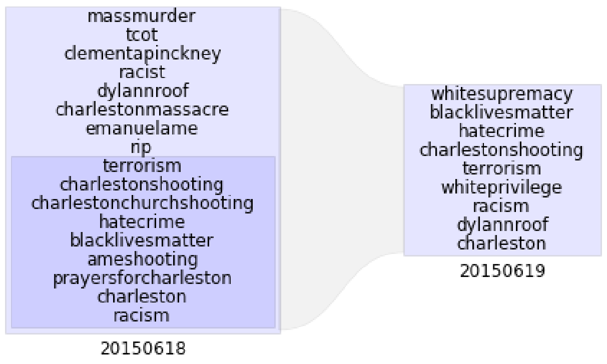

Figure 7: A cluster from one day can be related to a cluster on the next day with likelihood proportional to their shared elements

This transition matrix describes the similarity between a cluster in one time-period (rows) and a corresponding cluster in the subsequent time-period (columns). If a cluster remains unchanged from one time-period to the next, the value at the location in the transition matrix with row index corresponding to the cluster on the first day and column index corresponding to the cluster on the second day will be equal to 1. On the other hand, if the cluster breaks into two equal and fully separate groups, each of these will have a value of .5 in their corresponding column, as for cluster 20150618-cl2 in the table. When one of the groups is larger than the other, it will be given a correspondingly higher value. If two overlapping clusters form by splitting the original cluster, their combined values may exceed 1, as for cluster 20150618-cl3, and if significant fractions of the original cluster are not present in any subsequent cluster, the sum of all values for that row may be less than 1.

We should thus interpret the transition matrix not as a way to select the single subsequent cluster out of many which represents the original cluster

Table 1: Clusters at t1 have some likelihood of continuing as clusters at t2

Cluster ID	20150619-cl1	20150619-cl2	20150619-cl3	20150619-cl4	...
20150618-cl1	**0.37**	0.02	0.0	...	...
20150618-cl2	...	0.5	0.5	...	...
20150618-cl3	...	...	0.65	0.75	...
...	...	...	...	...	...

transformed and brought forward in time. Instead the value should be interpreted as the confidence that a subsequent cluster may be considered a continuation of the original cluster. Thus by looking at the distribution of values within a row in the transition matrix, we can identify situations in which a conversation splits, or multiple conversations merge to form a single larger conversation.

These dynamics point to underlying processes of polarization, factionalism, or reconciliation within the population engaging in the discussion. We should expect that in the process of meaning-making with regards to an event, the emergence of consensus should be characterized as a move toward fewer clusters as some clusters merge and others fall out of the larger discussion. Increasing polarization suggests the fission of conversations into partially overlapping clusters, as individuals from each faction begin to talk past one another, highlighting different aspects of the overarching conversation.

We can extend nested cluster diagram shown in Figure 5 to illustrate how

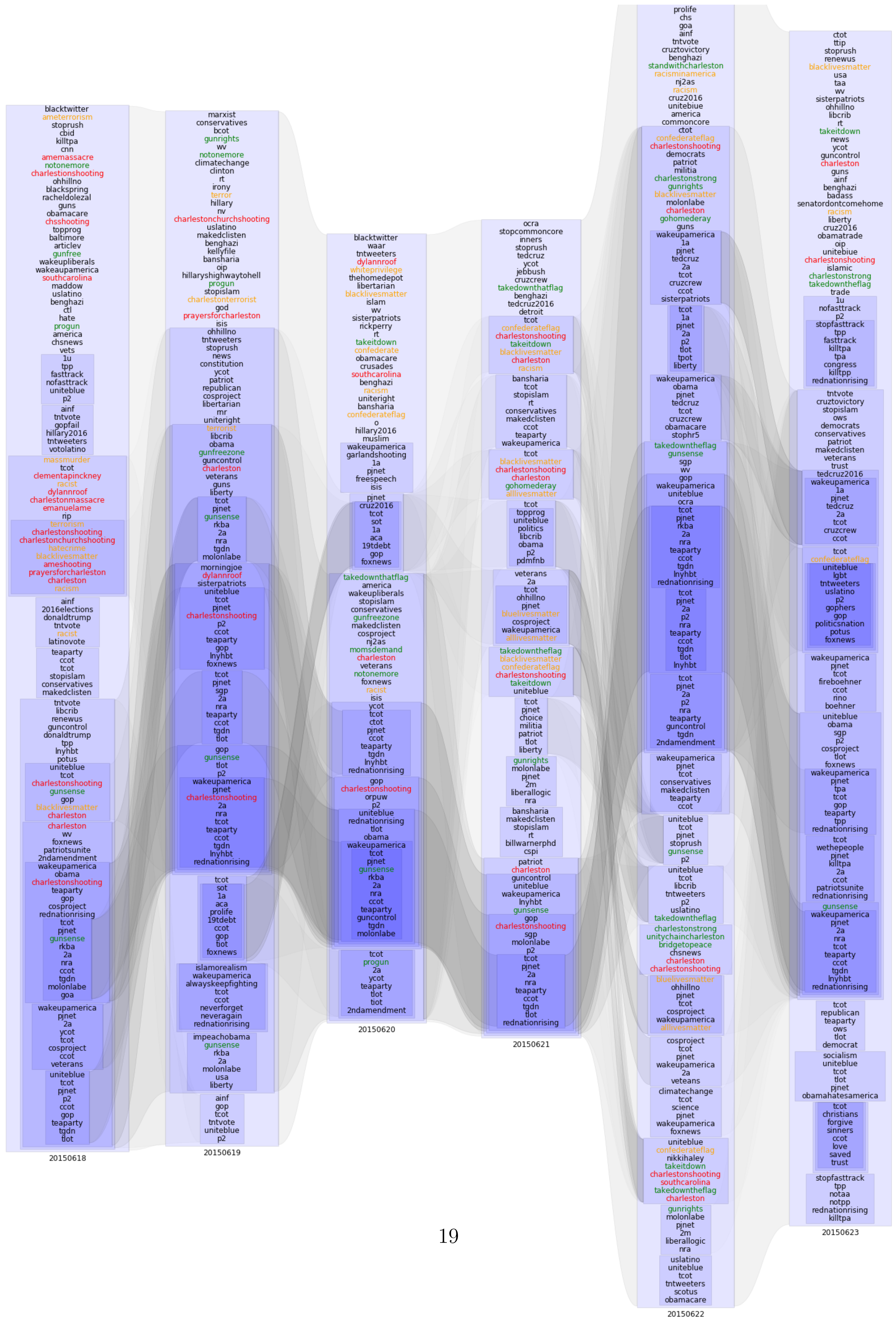

Figure 8: Weighted traces connect conversational clusters as they evolve day to day

clusters transition from day to day. In Figure 8, weighted traces connect conversations on June 18th to their likely continuations on June 19th, then 20th, and so on, with heavier traces implying greater continuity from day to day.

In the first column of this figure, corresponding to June 18, 2015, we see the conversation illustrated by Figure 5 in the context of a larger conversation also including preexisting clusters of conservative politics, the 2016 election cycle, and issues such as gun control and the Trans Pacific Partnership. In this first day following the shooting, a tight conversational cluster relates the details of the shooting: the event and its location (‘charlestonshooting’, ‘massmurder’, etc.), the victims (‘clementapinkney’, ‘emanuelame’), and the perpetrator (‘dylannroof’). These we highlight in red. Using the language of [Benford and Snow, 2000], we also see words representing early ‘diagnostic framing’; which define the event as a problem, explain the cause of that problem, and attribute blame. These we highlight in orange (‘racism’, ‘terrorism’).

In the second column, corresponding to June 19th, concrete details of the shooting no longer form a tight cluster, but references to the event (‘charlestonshooting’) become more central to the conversation regarding conservative politics, suggesting that a process of meaning-making is underway. References to existing ‘prognostic’ framings, which articulate appropriate response strategies, likewise move more towards the core of the political discussion as the (sadly) familiar responses to gun violence are brought out.

These we highlight in green (‘progun’, ‘gunfreezone’, ‘notonemore’).

In June 20, additional references to existing prognostic framings move into the core of the conversation (‘momsdemand’), and new diagnostic framings enter the larger conversation as details of the perpetrator’s motivation emerge (‘confederate’, ‘whiteprivilege’, ‘confederateflag’). As calls are made for the South Carolina state legislature to remove the Confederate flag from the state capitol grounds, new prognostic framings enter the discussion (‘takeitdown’, ‘takedownthatflag’). In the fourth column, representing June 21st, diagnostic and prognostic framings gel together in clusters linking the Confederate flag to the Charleston shooting and racism, and to the Black Lives Matter movement. A counterframing (‘alllivesmatter’, ‘gohomederay’) forms its own overlapping conversational cluster.

On June 22, the shooting itself has become less tightly linked with political discussion, as it is replaced with prognostic framings and calls to solidarity actions (‘charlestonstrong’, ‘unitychaincharleston’, ‘bridgetopeace’). In response to political pressure, the Confederate flag is removed from the state capitol. On June 23rd, the tight clustering of prognostic framing linked with the references to the shooting has dissipated, and the conversation returns to its longer-term structure, with movement references again moving toward the margins of the discussion.

Having explored the way the conversation that was formed in response to the shooting evolved into a conversation about the state’s support for symbols of racism, we can finally return to the original task of assessing public engagement with the discussion. In figure 9 we show the volume associated with each of the sub-conversations and some of the key terms associated with their evolution. Here we observe that in shifting its emphasis from a conversation about the event alone to a discussion with more political relevance, overall engagement with the topic actually surpasses the original discussion about the event on the first day.

This leads to the sociological conclusion that rather than the typical news cycle, what we see in this case is actually the trigger of a small wave of social activism linked to a larger social movement.

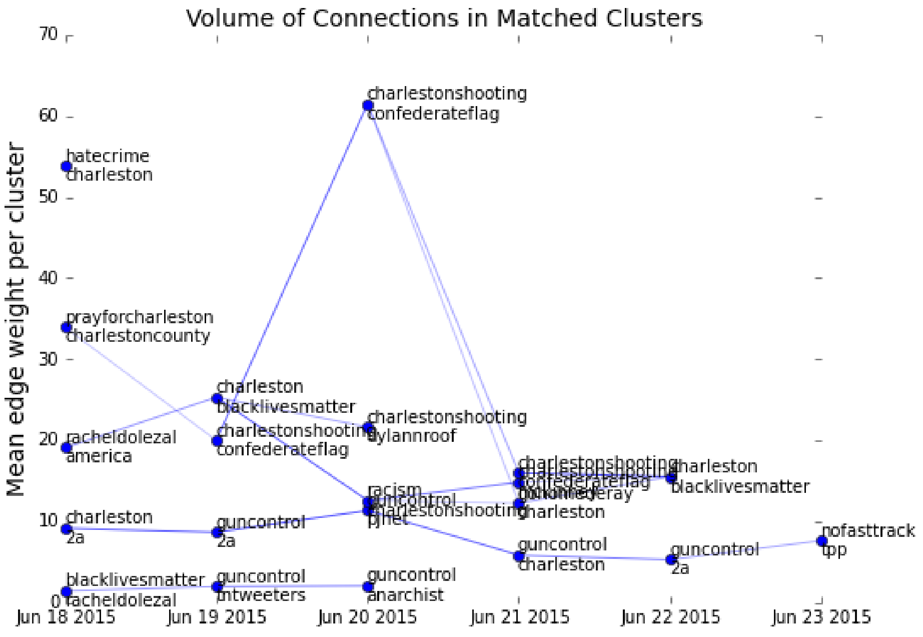

Figure 9: The average (1% sample) volume of messages in each evolving cluster shows how engagement with the conversation (as opposed to specific keywords) varies over time

5 Discussion and Conclusion

Through our analysis, it is clear that the 'National Conversation on Racism' that gained steam following the Charleston shooting was not a single discussion of a single set of issues. By structuring as a semantic network the way various concepts built and reflected on one another, we recognized that there may have been be multiple simultaneous but separate conversations addressing different aspects of the tragedy. By examining the density of relationships within each conversation, we found the central aspects of each conversation. By tracking the evolution of these conversations over time, we saw how events became linked to larger social issues, and how these links drew on prior discussion to frame and suggest responses to the events. These conclusions can only be drawn by understanding the links between the event and its interpretations as they evolve over time.

The utility of social media analysis for sociological research can be extended well beyond the practice of tracking keyword volume, net post sentiment, or supervised classification. In particular, tracking conversational clusters within a network of idea co-occurrences can give both structural understanding of a conversation, and insight into how it develops over time. These tools can prove helpful to sociologists interested in using social media to understand how world events are framed within the context of existing conversations.

Follow-on work to this paper could attempt to separate the various conversational clusters according to the groups engaged in them, possibly using information about the Twitter social network to understand if certain conversations propagate through topologically separate subgraphs, or if multiple conversations occur simultaneously within the same interconnected communities. Such research would have obvious impact on our understanding of framing, polarization, and the formation of group values.

6 Author's Note

The appendices to this paper contain all of the code needed to collect necessary data, generate the analysis, and and produce visualizations found herein:

Appendix A Cluster Identification and Transition Analysis in Python

Appendix B Cluster Identification and Transition Analysis in Unicage

Appendix C Performance Comparison Between Python and Unicage Examples

Appendix D Data Collection Scripts

Appendix E Visualizations

Appendix F Comparison of hashtags-only analysis with full analysis

The full set of scripts, and associated documentation, can be found at `https://github.com/JamesPHoughton/twitter-cluster`.

References

[Agarwal et al., 2014] Agarwal, S. D., Bennett, W. L., Johnson, C. N., and Walker, S. (2014). A model of crowd enabled organization: Theory and methods for understanding the role of twitter in the occupy protests. *International Journal of Communication*, 8:646–672.

[Bail, 2015] Bail, C. A. (2015). Taming Big Data: Using App Technology to Study Organizational Behavior on Social Media. *Sociological Methods & Research*.

[Becker et al., 2011] Becker, H., Naaman, M., and Gravano, L. (2011). Beyond Trending Topics: Real-World Event Identification on Twitter. *ICWSM*.

[Benford and Snow, 2000] Benford, R. D. and Snow, D. a. (2000). Framing Processes and Social Movements : An Overview and Assessment. *Annual Review Sociology*, 26(1974):611–639.

[Cogan and Andrews, 2012] Cogan, P. and Andrews, M. (2012). Reconstruction and analysis of twitter conversation graphs. *Proceedings of the First ACM International Workshop on Hot Topics on Interdisciplinary Social Networks Research*.

[Collins and Quillian, 1969] Collins, A. M. and Quillian, M. R. (1969). Retrieval Time from Semantic Memory. *Journal of Verbal Learning and Verbal Behavior*, 8:240–247.

[DiGrazia, 2015] DiGrazia, J. (2015). Using Internet Search Data to Produce State-level Measures: The Case of Tea Party Mobilization. *Sociological Methods & Research*.

[Gregori et al., 2013] Gregori, E., Lenzini, L., and Mainardi, S. (2013). Parallel k-clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems*, 24(8).

[Huberty, 2013] Huberty, M. (2013). Multi-cycle forecasting of congressional elections with social media. *Proceedings of the 2nd workshop on Politics, Elections and Data*.

[Marres and Gerlitz, 2014] Marres, N. and Gerlitz, C. (2014). Interface Methods: Renegotiating relations between digital research, STS and Sociology.

[Mayer, 1995] Mayer, R. E. (1995). The Search for Insight: Grappling with Gestalt Psychology's Unanswered Questions. In Sternberg, R. J. and Davidson, J. E., editors, *The Nature of Insight*, pages 1 online resource (xviii, 618 p.)–1 online resourc.

[McCormick et al., 2015] McCormick, T. H., Lee, H., Cesare, N., Shojaie, A., and Spiro, E. S. (2015). Using Twitter for Demographic and Social Science Research: Tools for Data Collection and Processing. *Sociological Methods & Research*.

[Morales et al., 2015] Morales, A. J., Borondo, J., Losada, J. C., and Benito, R. M. (2015). Measuring Political Polarization: Twitter shows the two sides of Venezuela. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 25(3).

[Palla et al., 2007] Palla, G., Barabási, A., and Vicsek, T. (2007). Quantifying social group evolution. *Nature*.

[Porter et al., 2009] Porter, M. a., Onnela, J.-P., and Mucha, P. J. (2009). Communities in Networks. *American Mathematical Society*, 56(9):0–26.

[Ritter et al., 2010] Ritter, A., Cherry, C., and Dolan, B. (2010). Unsupervised modeling of twitter conversations. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the ACL*, pages 172–180.

[Schilling, 2005] Schilling, M. A. (2005). A "Small-World" Network Model of Cognitive Insight. *Creativity Research Journal*, 17(2-3):131–154.

[Smith and Rainie, 2014] Smith, M. and Rainie, L. (2014). Mapping twitter topic networks: From polarized crowds to community clusters.

[Tounaka, 2013] Tounaka, N. (2013). How to Analyze 50 Billion Records in Less than a Second without Hadoop or Big Iron.

[Tulving, 1972] Tulving, E. (1972). Episodic and semantic memory.

[Tumasjan et al., 2010] Tumasjan, A., Sprenger, T., Sandner, P., and Welpe, I. (2010). Predicting Elections with Twitter: What 140 Characters Reveal about Political Sentiment. *ICWSM*.

[Woods, 1975] Woods, W. A. (1975). WHAT'S IN A LINK: Foundations for Semantic Networks. Technical Report November.

[Zachary, 1977] Zachary, W. W. (1977). An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33(4):452–473.

[Zubiaga et al., 2011] Zubiaga, A., Spina, D., Fresno, V., and Martínez, R. (2011). Classifying trending topics: a typology of conversation triggers on twitter. *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2461–2464.

Appendix A: Cluster Identification and Transition Analysis in Python

This code takes messages that are on twitter, and extracts their hashtags. It then constructs a set of weighted and unweighted network structures based upon co-citation of hashtags within a tweet. The network diagrams are interpreted to have a set of clusters within them which represent 'conversations' that are happening in the pool of twitter messages. We track similarity between clusters from day to day to investigate how conversations develop.

Utilities

These scripts depend upon a number of external utilities as listed below:

```python
import datetime
print 'started at %s'%datetime.datetime.now()
```

```python
import json
import gzip
from collections import Counter
from itertools import combinations
import glob
import dateutil.parser
import pandas as pd
import os
import numpy as np
import datetime
import pickle
import subprocess
```

```python
#load the locations of the various elements of the analysis
with open('config.json','r') as jfile:
    config = json.load(jfile)
print config
```

Data Files

We have twitter messages saved as compressed files, where each line in the file is the JSON object that the twitter sample stream returns to us. The files are created by splitting the streaming dataset according to a fixed number of lines - not necessarily by a fixed time or date range. A description of the collection process can be found in Appendix D.

All the files have the format `posts_sample_YYYYMMDD_HHMMSS_aa.txt` where the date listed is the date at which the stream was initialized. Multiple days worth of stream may be grouped under the same second, as long as the stream remains unbroken. If we have to restart the stream, then a new datetime will be added to the files.

```
# Collect a list of all the filenames that will be working with
files = glob.glob(config['data_dir']+'posts_sample*.gz')
print 'working with %i input files'%len(files)
```

Supporting Structures

Its helpful to have a list of the dates in the range that we'll be looking at, because we can't always just add one to get to the next date. Here we create a list of strings with dates in the format 'YYYYMMDD'. The resulting list looks like:

```
['20141101', '20141102', '20141103', ... '20150629', '20150630']
```

```
dt = datetime.datetime(2014, 11, 1)
end = datetime.datetime(2015, 7, 1)
step = datetime.timedelta(days=1)

dates = []
while dt < end:
    dates.append(dt.strftime('%Y%m%d'))
    dt += step

print 'investigating %i dates'%len(dates)
```

Step 1: Count hashtag pairs

The most data-intensive part of the analysis is this first piece, which parses all of the input files, and counts the various combinations of hashtags on each day.

In this demonstration we perform this counting in memory, which is sufficient for date ranges on the order of weeks, but becomes unwieldy beyond this timescale.

```
#construct a counter object for each date
tallydict = dict([(date, Counter()) for date in dates])

#iterate through each of the input files in the date range
for i, zfile in enumerate(files):
    if i%10 == 0: #save every 10 files
        print i,
        with open(config['python_working_dir']+"tallydict.pickle", "wb" ) as picklefile:
            pickle.dump(tallydict, picklefile)
        with open(config['python_working_dir']+"progress.txt", 'a') as progressfile:
            progressfile.write(str(i)+': '+zfile+'\n')

    try:
        with gzip.open(zfile) as gzf:
            #look at each line in the file
            for line in gzf:
                try:
                    #parse the json object
                    parsed_json = json.loads(line)
                    # we only want to look at tweets that are in
                    # english, so check that this is the case.
                    if parsed_json.has_key('lang'):
                        if parsed_json['lang'] =='en':
                            #look only at messages with more than two hashtags,
                            #as these are the only ones that make connections
                            if len(parsed_json['entities']['hashtags']) >=2:
                                #extract the hashtags to a list
                                taglist = [entry['text'].lower() for entry in
                                           parsed_json['entities']['hashtags']]
                                # identify the date in the message
                                # this is important because sometimes messages
                                # come out of order.
                                date = dateutil.parser.parse(parsed_json['created_at'])
                                date = date.strftime("%Y%m%d")
                                #look at all the combinations of hashtags in the set
                                for pair in combinations(taglist, 2):
                                    #count up the number of alpha sorted tag pairs
                                    tallydict[date][' '.join(sorted(pair))] += 1
                except: #error reading the line
                    print 'd',
    except: #error reading the file
        print 'error in', zfile
```

We save the counter object periodically in case of a serious error. If we have one, we can load what we've already accomplished with the following:

```
with open(config['python_working_dir']+"tallydict.pickle", "r" ) as picklefile:
    tallydict = pickle.load(picklefile)

print 'Step 1 Complete at %s'%datetime.datetime.now()
```

Step 2: Create Weighted Edge Lists

Having created this sorted set of tag pairs, we should write these counts to files. We'll create one file for each day. The files themselves will have one pair of words followed by the number of times those hashtags were spotted in combination on each day. For Example:

```
PCMS champs 3
TeamFairyRose TeamFollowBack 3
instadaily latepost 2
LifeGoals happy 2
DanielaPadillaHoopsForHope TeamBiogesic 2
shoes shopping 5
kordon saatc 3
DID Leg 3
entrepreneur grow 11
Authors Spangaloo 2
```

We'll save these in a very specific directory structure that will simplify keeping track of our data down the road, when we want to do more complex things with it. An example:

```
twitter/
   20141116/
       weighted_edges_20141116.txt
   20141117/
       weighted_edges_20141117.txt
   20141118/
       weighted_edges_20141118.txt
   etc...
```

We create a row for every combination that has a count of at least two.

In this code we'll use some of the iPython 'magic' functions for file manipulation, which let us execute shell commands as if through a terminal. Lines prepended with the exclaimation point `!` will get passed to the shell. We can include python variables in the command by prepending them with a dollar sign `$`.

```
for key in tallydict.keys(): #keys are datestamps
    #create a directory for the date in question
    date_dir = config['python_working_dir']+key
    if not os.path.exists(date_dir):
        os.makedirs(date_dir)
    #replace old file, instead of append
    with open(config['python_working_dir']+key+'/weighted_edges_'+key+'.txt', 'w') as fout:
        for item in tallydict[key].iteritems():
            if item[1] >= 2: #throw out the ones that only have one edge
                fout.write(item[0].encode('utf8')+' '+str(item[1])+'\n')
```

Now lets get a list of the wieghed edgelist files, which will be helpful later on.

```
weighted_files = glob.glob(config['python_working_dir']+'*/weight*.txt')
print 'created %i weighted edgelist files'%len(weighted_files)
print 'Step 2 Complete at %s'%datetime.datetime.now()
```

Step 3: Construct unweigheted edgelist

We make an unweighted list of edges by throwing out everything below a certain threshold. We'll do this for a range of different thresholds, so that we can compare the results later. Looks like:

```
FoxNflSunday tvtag
android free
AZCardinals Lions
usa xxx
كبلز متحرره
CAORU TEAMANGELS
RT win
FarCry4 Games
```

We do this for thresholds between 2 and 15 (for now, although we may want to change later) so the directory structure looks like:

`twitter/`
`20141116/`
`th_02/`

```
unweighted_20141116_th_02.txt
th_03/
unweighted_20141116_th_03.txt
th_04/
unweighted_20141116_th_04.txt
etc...
20151117/
th_02/
unweighted_20141117_th_02.txt
etc...
etc...
```

Filenames include the date and the threshold, and the fact that these files are unweighted edge lists.

```
for threshold in range (2, 15):
    for infile_name in weighted_files:
        date_dir = os.path.dirname(infile_name)
        date = date_dir.split('/')[-1]
        weighted_edgefile = os.path.basename(infile_name)

        #create a subdirectory for each threshold we choose
        th_dir = date_dir+'/th_%02i'%threshold
        if not os.path.exists(th_dir):
            os.makedirs(th_dir)

        # load the weighted edgelists file and filter it to
        # only include values above the threshold
        df = pd.read_csv(infile_name, sep=' ', header=None,
                          names=['Tag1', 'Tag2', 'count'])
        filtered = df[df['count']>threshold][['Tag1','Tag2']]

        #write out an unweighted edgelist file for each threshold
        outfile_name = th_dir+'/unweighted_'+date+'_th_%02i'%threshold+'.txt'
        with open(outfile_name, 'w') as fout: #replace old file, instead of append
            for index, row in filtered.iterrows():
                try:
                    fout.write(row['Tag1']+' '+row['Tag2']+'\n')
                except:
                    print 'b',
```

Now lets get a list of all the unweighted edgelist files we created

```
unweighted_files = glob.glob(config['python_working_dir']+'*/*/unweight*.txt')
print 'created %i unweighted edgelist files'%len(unweighted_files)
print 'Step 3 Complete at %s'%datetime.datetime.now()
```

Step 4: Find the communities

We're using COS Parallel to identify k-cliques, so we feed each unweighted edge file into the `./maximal_cliques` preprocessor, and then the `./cos` algorithm.

The unweighed edgelist files should be in the correct format for `./maximal_cliques` to process at this point.

`./maximal_cliques` translates each node name into an integer to make it faster and easier to deal with, and so the output from this file is both a listing of all of the maximal cliques in the network, with an extension `.mcliques`, and a mapping of all of the integer nodenames back to the original text names, having extension `.map`.

It is a relatively simple task to feed each unweighed edgelist we generated above into the `./maximal_cliques` algorithm.

```
for infile in unweighted_files:
    th_dir = os.path.dirname(infile)
    th_file = os.path.basename(infile)
    #operate the command in the directory where we want the files created
    subprocess.call([os.getcwd()+'/'+config['maximal_cliques'], th_file], cwd=th_dir)
```

Step 5: Once this step is complete, we then feed the `.mcliques` output files into the `cosparallel` algorith.

```
maxclique_files = glob.glob(config['python_working_dir']+'*/*/*.mcliques')
print 'created %i maxcliques files'%len(maxclique_files)
print 'Step 4a Complete at %s'%datetime.datetime.now()
```

```
current_directory = os.getcwd()
for infile in maxclique_files:
    mc_dir = os.path.dirname(infile)
    mc_file = os.path.basename(infile)
    subprocess.call([os.getcwd()+'/'+config['cos-parallel'], mc_file], cwd=mc_dir)
```

```
community_files = glob.glob(config['python_working_dir']+'*/*/[0-9]*communities.txt')
print 'created %i community files'%len(community_files)
print 'Step 5 Complete at %s'%datetime.datetime.now()
```

Step 6: Translate back from numbers to actual words

The algorithms we just ran abstract away from the actual text words and give us a result with integer collections and a map back to the original text. So we apply the map to recover the clusters in terms of their original words, and give each cluster a unique identifier:

```
0 Ferguson Anonymous HoodsOff OpKKK
1 Beauty Deals Skin Hair
2 Family gym sauna selfie
etc...
```

```python
# we'll be reading a lot of files like this,
# so it makes sense to create a function to help with it.
def read_cluster_file(infile_name):
    """ take a file output from COS and return a dictionary
    with keys being the integer cluster name, and
    elements being a set of the keywords in that cluster"""
    clusters = dict()
    with open(infile_name, 'r') as fin:
        for i, line in enumerate(fin):
            #the name of the cluster is the bit before the colon
            name = line.split(':')[0]
            if not clusters.has_key(name):
                clusters[name] = set()
            #the elements of the cluster are after the colon, space delimited
            nodes = line.split(':')[1].split(' ')[:-1]
            for node in nodes:
                clusters[name].add(int(node))
    return clusters


current_directory = os.getcwd()
for infile in community_files:
    c_dir = os.path.dirname(infile)
    c_file = os.path.basename(infile)

    #load the map into a pandas series to make it easy to translate
    map_filename = glob.glob('%s/*.map'%c_dir)
    mapping = pd.read_csv(map_filename[0], sep=' ', header=None,
                          names=['word', 'number'], index_col='number')

    clusters = read_cluster_file(infile)
    #create a named cluster file in the same directory
    with open(c_dir+'/named'+c_file, 'w') as fout:
        for name, nodes in clusters.iteritems():
            fout.write(' '.join([str(name)]+
                                [mapping.loc[int(node)]['word'] for node in list(nodes)]+
                                ['\n']))
```

```python
print 'Step 6 Complete at %s'%datetime.datetime.now()
```

While we're at it, we'll write a function to read the files we're creating

```
def read_named_cluster_file(infile_name):
    """ take a file output from COS and return a """
    clusters = dict()
    with open(infile_name, 'r') as fin:
        for i, line in enumerate(fin):
            name = line.split(' ')[0]
            if not clusters.has_key(name):
                clusters[int(name)] = set()
            nodes = line.split(' ')[1:-1]
            for node in nodes:
                clusters[int(name)].add(node)
    return clusters
```

Step 7: Compute transition likelihoods

We want to know how a cluster on one day is related to a cluster on the next day. For now, we'll use a brute-force algorithm of counting the number of nodes in a cluster that are present in each of the subsequent day's cluster. From this we can get a likelihood of sorts for subsequent clusters.

We'll define a function that, given the clusers on day 1 and day 2, creates a matrix from the two, with day1 clusters as row elements and day2 clusters as column elements. The entries to the matrix are the number of nodes shared by each cluster.

```
#brute force, without the intra-day clustering
def compute_transition_likelihood(current_clusters, next_clusters):
    transition_likelihood = np.empty([max(current_clusters.keys())+1,
                                      max(next_clusters.keys())+1])
    for current_cluster, current_elements in current_clusters.iteritems():
        for next_cluster, next_elements in next_clusters.iteritems():
            #the size of the intersection of the sets
            transition_likelihood[current_cluster, next_cluster] = (
                               len(current_elements & next_elements) )
    return transition_likelihood
```

We want to compute transition matricies for all clusters with every k and every threshold. We'll save the matrix for transitioning from Day1 to Day2 in Day1's folder. In many cases, there won't be an appropriate date/threshold/k combination, so we'll just skip that case.

```
#this should compute and store all of the transition likelihoods

for current_date in dates[:-1]:
    next_date = dates[dates.index(current_date)+1]
    for threshold in range(2,15):
        for k in range(3, 20):

            current_file_name = config['python_working_dir']+'%s/th_%02i/named%i_communities.t
                                                                  threshold, k)

            next_file_name = config['python_working_dir']+'%s/th_%02i/named%i_communities.txt'
                                                               threshold, k)

            if os.path.isfile(current_file_name) & os.path.isfile(next_file_name):
                current_clusters = read_named_cluster_file(current_file_name)
                next_clusters = read_named_cluster_file(next_file_name)

                transition = compute_transition_likelihood(current_clusters,
                                                           next_clusters)
                transitiondf = pd.DataFrame(data=transition,
                                            index=current_clusters.keys(),
                                            columns=next_clusters.keys())

                transitiondf.to_csv(current_file_name[:-4]+'_transition.csv')
```

```
transition_files = glob.glob(config['python_working_dir']+'*/*/named*_communities_transition.c
print 'created %i transition matrix files'%len(transition_files)
print 'Step 6 Complete at %s'%datetime.datetime.now()
```

```

```

Appendix B: Cluster Identification and Transition Analysis in Unicage

This code replicates the functionality found in appendix A one step at a time, using shell programming and the Unicage development platform. Each of the scripts listed here is found at https:\github.com\ **Removed for Anonymity**

Running these scripts

This analysys process is separated to 7 steps. You can run each or all steps using the helper script `twitter_analysis.sh` as follows:

```
$ twitter_analysis.sh <start_step_no> <end_step_no>
```

A key to the step numbers is:

```
1 - list_word_pairings.sh
2 - wgted_edge_gen.sh
3 - unwgted_edge_gen.sh
4 - run_mcliques.sh
5 - run_cos.sh
6 - back_to_org_words.sh
7 - compute_transition_likelihoods.sh
```

For example, to execute step4 to step6:

```
shell
$ twitter_analysis.sh 4 6
```

To execute step2 only:

```
shell
$ twitter_analysis.sh 2 2
```

To execute all steps:

```
shell
$ twitter_analysis.sh 1 7
```

Step 1: `list_word_pairings.sh`

This script creates lists of hashtag pairs from json files.

Output: produces `DATA/result.XXXX.`

```
#!/bin/bash

homed=/home/James.P.H/UNICAGE
toold=${homed}/TOOL
shelld=${homed}/SHELL
rawd=/home/James.P.H/data
semd=${homed}/SEMAPHORE
datad=${homed}/DATA
workd=${homed}/twitter

tmp=/tmp/$$

mkdir -p ${datad}

n=0

# Process zipped files/
echo ${rawd}/posts_sample*.gz                                           |
tarr                                                                    |
while read zipfile; do
  n=$((n+1))
  echo $zipfile $n

  {
    zcat $zipfile                                                       |
    ${homed}/SHELL/myjsonparser |
    # 1: "time" 2: timestamp (epoch msec) 3: "hashtag" 4-N: hashtags

    awk 'NF>5{for(i=4;i<=NF;i++)for(j=i+1;j<=NF;j++){print $i,$j,int($2/1000)}}' |
    # list all possible 2 word combinations with timestamp. 1: word1 2: word2 3: timestamp (epo

    TZ=UTC calclock -r 3                                                       |
    # 1: word1 2: word2 3: timestamp (epoch sec) 4: timestamp (YYYYMMDDhhmmss)

    self 1 2 4.1.8                                                      |
    # 1: word1 2: word2 3: timestamp (YYYYMMDD)

    msort key=1/3                                                       |
    count 1 3                                                       > ${datad}/result.$n
    # count lines having the same word combination and timestamp 1:word1 2:word2 3:date 4:coun

    # run 5 processes in parallel
    touch ${semd}/sem.$n
   } &
   if [ $((n % 5)) -eq 0 ]; then
     eval semwait ${semd}/sem.{$((n-4))..$n}
     eval rm -f ${semd}/sem.* 2> /dev/null
   fi
done

wait

n=$(ls ${datad}/result.* | sed -e 's/\./ /g' | self NF | msort key=1n | tail -1)

# Process unzipped files.
#    *There are unzipped files in raw data dir(/home/James.P.H/data).
echo ${rawd}/posts_sample*                      |
tarr                                    |
self 1 1.-3.3                               |
delr 2 '.gz'                                |
self 1                                  |
while read nozipfile; do
  n=$((n+1))
  echo $nozipfile $n

  {
    cat $nozipfile                                                  |
    ${homed}/SHELL/myjsonparser |
    # 1: "time" 2: timestamp (epoch msec) 3: "hashtag" 4-N: hashtags

    awk 'NF>5{for(i=4;i<=NF;i++)for(j=i+1;j<=NF;j++){print $i,$j,int($2/1000)}}' |
    # list all possible 2 word combinations with timestamp. 1: word1 2: word2 3: timestamp (epo

    calclock -r 3                                                   |
    # 1: word1 2: word2 3: timestamp (epoch sec) 4: timestamp (YYYYMMDDhhmmss)

    self 1 2 4.1.8                                                  |
    # 1: word1 2: word2 3: timestamp (YYYYMMDD)

    msort key=1/3                                                   |
    count 1 3                                                       > ${datad}/result.$n
    # count lines having the same word combination and timestamp 1:word1 2:word2 3:date 4:coun
    # run 5 processes in parallel
    touch ${semd}/sem.$n
   } &
   if [ $((n % 5)) -eq 0 ]; then
     eval semwait ${semd}/sem.{$((n-4))..$n}
     eval rm -f ${semd}/sem.* 2> /dev/null
   fi
done

#semwait "${semd}/sem.*"
wait
eval rm -f ${semd}/sem.* 2> /dev/null

rm -f $tmp-*

exit 0
```

Step 2: `wgted_edge_gen.sh`

This script creates weighted edgelists from `result.*` and places them under yyyymmdd dirs.

Output: produces `twitter/yyyymmdd/weighted_edges_yyyymmdd.txt`

```
#!/bin/bash -xv

# wgted_edge_gen.sh creates weighted edgelists from result.*
# and place them under yyyymmdd dirs.

homed=/home/James.P.H/UNICAGE
toold=${homed}/TOOL
shelld=${homed}/SHELL
rawd=/home/James.P.H/data
semd=/${homed}/SEMAPHORE
datad=${homed}/DATA
workd=${homed}/twitter

# TODO debug
#datad=${homed}/DATA.mini
#workd=${homed}/twitter.mini


tmp=/tmp/$$
```

```
# error function: show ERROR and exit with 1
ERROR_EXIT() {
  echo "ERROR"
  exit 1
}

mkdir -p ${workd}

# count the number of files
n=$(ls ${datad}/result.* | gyo)

for i in $(seq 1 ${n} | tarr)
do
    # 1:Tag1 2:Tag2 3:date 4:count
    sorter -d ${tmp}-weighted_edges_%3_${i} ${datad}/result.${i}
    [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT
done

# listup target dates
echo ${tmp}-weighted_edges_????????_*      |
tarr                                       |
ugrep -v '\?'                              |
sed -e 's/_/ /g'                           |
self NF-1                                  |
msort key=1                                |
uniq                    > ${tmp}-datelist
# 1:date(YYYYMMDD)

[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

for day in $(cat ${tmp}-datelist); do
  mkdir -p ${workd}/${day}

  cat ${tmp}-weighted_edges_${day}_*    |
  # 1:word1 2:word2 3:count
  msort key=1/2             |
  sm2 1 2 3 3               > ${workd}/${day}/weighted_edges_${day}.txt
  # 1:word1 2:word2 3:count

  [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

done

rm ${tmp}-*
```

```
exit 0
```

Step 3: `unwgted_edge_gen.sh`

This script creates unweighted edgelists under the same dir sorted by threshold dirs.

Output: produces `twitter/yyyymmdd/th_XX/unweighted_yyyymmdd_th_XX.txt`

```
#!/bin/bash -xv

# unwgted_edge_gen.sh expects weighted edgelists
# (weighted_edges_yyyymmdd.txt) located in
# /home/James.P.H/UNICAGE/twitter/yyyymmdd
# and creates unweighted edgelists under the same dir
# sorted by threshold dirs.

homed=/home/James.P.H/UNICAGE
toold=${homed}/TOOL
shelld=${homed}/SHELL
rawd=/home/James.P.H/data
semd=${homed}/SEMAPHORE
datad=${homed}/DATA
workd=${homed}/twitter

# TODO test
#datad=${homed}/DATA.mini
#workd=${homed}/twitter.mini

tmp=/tmp/$$

# error function: show ERROR and delete tmp files
ERROR_EXIT() {
  echo "ERROR"
  rm -f $tmp-*
  exit 1
}

# setting threshold
seq 2 15 | maezero 1.2                         > $tmp-threshold
[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

# creating header file
itouch "Hashtag1 Hashtag2 count" $tmp-header
```

```
[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

# create list for all pairs of thresholds and filenames
echo ${workd}/201[45]*/weighted_edges_*.txt          |
tarr                                                 |
joinx $tmp-threshold -                               |
# 1:threshold 2:filename
while read th wgtedges ; do
   echo ${wgtedges}
   [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

   # define year-month-date variable for dir and file name
   yyyymmdd=$(echo ${wgtedges} | awk -F \/ '{print $(NF-1)}')
   [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

   echo ${yyyymmdd} th_${th}
   [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

   # create threshold dirs under twitter/YYYYMMDD
   mkdir -p $(dirname ${wgtedges})/th_${th}
   [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

   cat $tmp-header ${wgtedges}                       |
   # output lines whose count feild is above thresholds
   ${toold}/tagcond '%count > '"${th}"''             |
   # remove threshold feild
   tagself Hashtag1 Hashtag2                         |
   # remove header
   tail -n +2 > ${workd}/${yyyymmdd}/th_${th}/unweighted_${yyyymmdd}_th_${th}.txt
   [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

done
[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

# delete tmp files
rm -f $tmp-*

exit 0
```

Step 4: `run_mcliques.sh`

This script executes maximal_cliques to all unweigthed edges.

Output: produces
- `twitter/yyyymmdd/th_XX/unweighted_edges_yyyymmdd_th_XX.txt.map`
- `twitter/yyyymmdd/th_XX/unweighted_edges_yyyymmdd_th_XX.txt.mcliques`

```
#!/bin/bash -xv

# run_mcliques.sh executes maximal_cliques to all unweigthed edges.
# produce unweighted_edges_yyyymmdd.txt.map and unweighted_edges_yyyymmdd.txt.mcliques

homed=/home/James.P.H/UNICAGE
toold=${homed}/TOOL
shelld=${homed}/SHELL
rawd=/home/James.P.H/data
semd=${homed}/SEMAPHORE
datad=${homed}/DATA
workd=${homed}/twitter

# TODO test
#datad=${homed}/DATA.mini
#workd=${homed}/twitter.mini

# error function: show ERROR
ERROR_EXIT() {
echo "ERROR"
exit 1
}

# 共有ライブラリへパスを通す(maximal_cliques用)
LD_LIBRARY_PATH=/usr/local/lib:/usr/lib
export LD_LIBRARY_PATH


# running maximal_cliques
for unwgted_edges in ${workd}/*/th_*/unweighted_*_th_*.txt
do
    echo "Processing ${unwgted_edges}."
    [ $(plus $(echo ${PIPESTATUS[@]})) -eq "0" ] || ERROR_EXIT

    # skip empty files
    if [ ! -s ${unwgted_edges} ] ; then
        echo "Skipped $(basename ${unwgted_edges})."
        continue
    fi
    cd $(dirname ${unwgted_edges})
    [ $(plus $(echo ${PIPESTATUS[@]})) -eq "0" ] || ERROR_EXIT

    ${toold}/maximal_cliques ${unwgted_edges}
    [ $(plus $(echo ${PIPESTATUS[@]})) -eq "0" ] || ERROR_EXIT
    # unweighted_edges_yyyymmdd.txt.map (1:Tag 2:integer)
    # unweighted_edges_yyyymmdd.txt.mcliques (1...N: integer for nodes N+1: virtual node -1)

    echo "${unwgted_edges} done."
    [ $(plus $(echo ${PIPESTATUS[@]})) -eq "0" ] || ERROR_EXIT
done

exit 0
```

Step 5: `run_cos.sh`

This script executes `cos` using `*.mcliques` files to create communities.

Output: produces `twitter/yyyymmdd/th_XX/N_communities.txt`

```
#!/bin/bash -xv

# run_cos.sh creates communities using *.mcliques files.

homed=/home/James.P.H/UNICAGE
toold=${homed}/TOOL
shelld=${homed}/SHELL
rawd=/home/James.P.H/data
semd=${homed}/SEMAPHORE
datad=${homed}/DATA
workd=${homed}/twitter

# error function: show ERROR
ERROR_EXIT() {
echo "ERROR"
exit 1
}

# 共有ライブラリへパスを通す(cos用)
LD_LIBRARY_PATH=/usr/local/lib:/usr/lib
export LD_LIBRARY_PATH

# running cos
for mcliques in ${workd}/*/th_*/unweighted_*_th_*.txt.mcliques
do
    echo "Processing ${mcliques}."
    [ $(plus $(echo ${PIPESTATUS[@]})) -eq "0" ] || ERROR_EXIT

    # changing dir so that output files can be saved under each th dirs.
    cd $(dirname ${mcliques})
    [ $(plus $(echo ${PIPESTATUS[@]})) -eq "0" ] || ERROR_EXIT

    ${toold}/cos ${mcliques}
    [ $(plus $(echo ${PIPESTATUS[@]})) -eq "0" ] || ERROR_EXIT
    # N_communities.txt (1:community_id 2..N: maximal_clique)
    # k_num_communities.txt (1:k 2: number of k-clique communities discovered)

    echo "${mcliques} done."
    [ $(plus $(echo ${PIPESTATUS[@]})) -eq "0" ] || ERROR_EXIT
done

exit 0
```

Step 6: `back_to_org_words.sh`

This script reverts integers in `N_commnities.txt` to original words using map file generated by `maximal_cliques`.

Output: produces `twitter/yyyymmdd/th_XX/namedN_communities.txt`

```
#!/bin/bash -xv

# back_to_org_words.sh:
# use map file generated by maximal_cliques to revert integers in N_commnities.txt
# to original words.

homed=/home/James.P.H/UNICAGE
toold=${homed}/TOOL
shelld=${homed}/SHELL
rawd=/home/James.P.H/data
semd=${homed}/SEMAPHORE
datad=${homed}/DATA
workd=${homed}/twitter

tmp=/tmp/$$

# TODO test
#datad=${homed}/DATA.mini
#workd=${homed}/twitter.mini


# error function: show ERROR and delete tmp files
ERROR_EXIT() {
  echo "ERROR"
  rm -f $tmp-*
  exit 1
}


echo ${workd}/*/th_*/[0-9]*_communities.txt     |
tarr                            |
ugrep -v '\*'                       |

# community番号とthreshold数を変数に入れてwhileループをする必要がある

while read community_files; do
```

```
:>$tmp-tran

   echo ${community_files}

   # get directory path of target-file
   dirname=$(dirname ${community_files})
   # get filename
   filename=$(basename ${community_files})

   # read a community file
   fsed 's/:/ /1' ${community_files}                    |
   # 1: community id 2..N: integer for node

   # remove unnecessary space char at the end of each line
   sed -e 's/ *$//'                                     |

   # remove lines which have only 1 field for community id
   gawk 'NF>1'                    |

   tarr num=1                     |
   # 1: community id 2: integer

   self 2 1                     |
   # 1: integer 2: community id
   # sort by field 1/2
   msort key=1/2                                        |
   # remove the same records, only take last one
   getlast 1 2                                          > $tmp-tran
   # 1: integer 2: community id

   [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

   # TODO: for debug
   cat $tmp-tran

   # Read the word-map file
   cat ${dirname}/unweighted_*_th_*.txt.map             |
   # 1: word 2: integer
   self 2 1                                             |
   # 1: integer 2: word
   # sort by field 1
   msort key=1                                          |
   # join map file to community -tran
   join1 key=1 - $tmp-tran                              |
   # 1: integer 2: word 3: community id
```

```
    self 3 2                                          |
    # 1: community id 2: word
    yarr num=1 > ${dirname}/named${filename}
    # 1: community id 2..N: word1..N

    [ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

done
[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

# delete tmp files
rm -f $tmp-*

exit 0
```

Step 7: `compute_transition_likelihoods.sh`

This script will compute transition-likelihoods map files using `named_N_communities.txt`.

Output: produces `twitter/yyyymmdd/th_XX/namedN_communities_transition.csv`

```
#!/bin/bash -xv

# compute_transition_likelihoods.sh
#

homed=/home/James.P.H/UNICAGE
toold=${homed}/TOOL
shelld=${homed}/SHELL
rawd=/home/James.P.H/data
semd=${homed}/SEMAPHORE
datad=${homed}/DATA
workd=${homed}/twitter

tmp=/tmp/$$

# TODO test
#shelld=${homed}/SHELL/sugi_test
#datad=${homed}/DATA.mini
#workd=${homed}/twitter.mini

# error function: show ERROR and delete tmp files
```

```
ERROR_EXIT() {
  echo "ERROR"
  rm -f $tmp-*
  exit 1
}

# 対象の日付リストを作成
echo ${workd}/2*              |
tarr                          |
ugrep -v '\*'                 |
sed -e 's/\// /g'             |
self NF                       |
msort key=1                 > $tmp-date-dir-list
# 1:date(real dir)

[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

fromdate=$(head -1 $tmp-date-dir-list)
todate=$(tail -1 $tmp-date-dir-list)

mdate -e ${fromdate} ${todate}          > $tmp-date-list
# 1:date

echo ${workd}/20??????/th*/named*_communities.txt   |
tarr                                                |
grep -v '\*'                                        |
# 1:current_filename
self 1 1                                            |
fsed 's#/# #2'                                      |
# 1:current_filename ... NF-2:current_date NF-1:"th_XX" NF:"namedXX_communities.txt"
self 1 NF-2/NF                                      |
# 1:current_filename 2:current_date 3:"th_XX" 4:"namedXX_communities.txt"
mdate -f 2/+1                                       |
# 1:current_filename 2:current_date 3:next_date 4:"th_XX" 5:"namedXX_communities.txt"
gawk '{ print $1, "'${workd}'/"$3"/"$4"/"$5 }'      |
# 1:current_filename 2:next_filename
${shelld}/intersection
## output result to "${workd}/named*_communities_transition.work"" each line.

# create map file
echo ${workd}/20??????/th*/named*_communities_transition.work   |
tarr                                                            |
grep -v '\*'                                                    |
while read filename; do
      # csv file name
      csv_filename=$(echo $filename | sed 's/\./ /g' | self 1/NF-1 | sed 's/ /./g' | gawk '{ pr

      # create map: index=id(curr) columns=id(next)
      maezero 1.3 2.3 $filename     |
      map num=1x1 -                 |
      # comvert to csv
      tocsv             > ${csv_filename}
done


# delete tmp files
rm ${workd}/20??????/th*/named*_communities_transition.work
rm -f $tmp-*

exit 0
```

Appendix C: Performance Comparison

The scripts listed in appendices A and B were run on an 8 Core Intel 64 bit 3.2Ghz processor with 48gb ram, for messages over the date range

Step	**Python Runtime**	**Unicage Runtime**
1. Counting Hashtag Pairs	38h:34m:33s	06h:21m:34s
2. Weighted Edgelists	00h:00m:10s	00h:03m:31s
3. Unweighted Edgelists	00h:13m:36s	00h:01m:36s
4. Maximal Clique Identification	00h:10m:13s	00h:01m:32s
5. k-Clique Percolation	00h:05m:45s	00h:01m:42s
6. Mapping back to Text	00h:19m:56s	00h:04m:29s
7. Computing transition likelihoods	00h:03m:15s	00h:11m:03s
Total time	**39h:27m:28s**	**06h:45m:27s**

Appendix D: Data Collection Scripts

The following scripts are used to collect twitter messages and store them in a format accessible to both python and unicage. While a databasing system has its obvious advantages, this methodology is the paragon of simplicity.

Curl URL Builder

This script generates a properly signed URL for opening a twitter stream via curl

```
""" twitter_curl_url_builder.py """

import oauth2 as oauth
import time

# Set the API endpoint
url = 'https://stream.twitter.com/1.1/statuses/sample.json'

# Set the base oauth_* parameters along with any other parameters required
# for the API call.
params = {
    'oauth_version': "1.0",
    'oauth_nonce': oauth.generate_nonce(),
    'oauth_timestamp': int(time.time())
}

# Set up instances of our Token and Consumer.
token = oauth.Token(key='******************************',
                    secret='**************************')
consumer = oauth.Consumer(key='************************',
                          secret='********************')

# Set our token/key parameters
params['oauth_token'] = token.key
params['oauth_consumer_key'] = consumer.key

# Create our request. Change method, etc. accordingly.
req = oauth.Request(method="GET", url=url, parameters=params)

# Sign the request.
signature_method = oauth.SignatureMethod_HMAC_SHA1()
req.sign_request(signature_method, consumer, token)

print req.to_url()
```

To use, at the command prompt:

```
shell
$ URL=$(python twitter_curl_kickstarter.py)
$ curl -get "$URL"
```

Twitter Stream Opener

This script starts a curl process to get posts from twitter and saves them to 100000 post long files. We use the python script `twitter_curl_url_builder.py` to handle the oauth bits, as they can be complicated in bash.

```
#!/bin/bash

URL=$(python twitter_curl_url_builder.py)

curl --get "$URL" | split -l 100000 - ../data/posts_sample_`date "+%Y%m%d_%H%M%S"`_

echo "`date` Twitter stream broken with error: ${PIPESTATUS[0]}" >> tw_collect_log.txt

# the curl should go on indefinitely, so if we get to this point, an error has occurred, raise
exit 1
```

Twitter Stream Monitor

This script starts the stream, and watches to see if it fails. If so, it restarts the process after some amount of time.

For a great description of the watchdog loop, see: http://stackoverflow.com/questions/696839/how-do-i-write-a-bash-script-to-restart-a-process-if-it-dies

```
#!/bin/bash
reconnect_delay=600

until ./twitter_stream_opener.sh; do
    echo "`date` Twitter curl process interrupted. Attempting reconnect after $reconnect_delay

    echo "`date` Twitter curl process interrupted. Attempting reconnect after $reconnect_delay

    sleep "$reconnect_delay"

done
```

Appendix E: Visualization

Visualizations will use the following python librarires:

```
import pandas as pd
import glob
import datetime
import itertools
import matplotlib.pylab as plt
```

We define several helper functions to assist with reading the cluster files:

```
    def read_named_cluster_file(infile_name):
        """ take a file output from COS and return a dictionary,
        where keys are the name of a cluster,
        and values are sets containing names of nodes in the cluster"""
        clusters = dict()
        with open(infile_name, 'r') as fin:
            for i, line in enumerate(fin):
                name = line.split(' ')[0]
                if not clusters.has_key(name):
                    clusters[int(name)] = set()
                nodes = line.split(' ')[1:-1]
                for node in nodes:
                    clusters[int(name)].add(node)
        return clusters


    def get_clusters_with_keyword(date, threshold, keyword):
        """Get clusters from the dataset that include the keyword.
        Get them for the specified date and threshold.

        date : string in yyyymmdd format
        threshold : integer above 2

        """
        files = pd.DataFrame(glob.glob(date+'/th_%02i'%threshold+'/named*_communities.txt'), co
        files['clique_size']=files['filename'].apply(lambda x: int(x.split('named')[1].split('_
        files.sort('clique_size', ascending=False, inplace=True)
        outlist = []

        for index, row in files.iterrows():
            clusters = read_named_cluster_file(row['filename'])
            for index, cluster_set in clusters.iteritems():
                outdict = {}
                if keyword in cluster_set:
                    outdict['date'] = date
                    outdict['threshold'] = threshold
                    outdict['keyword'] = keyword
                    outdict['elements'] = cluster_set
                    outdict['k-clique'] = row['clique_size']
                    outdict['name'] = index
                    outdict['id'] = str(date)+'_k'+str(row['clique_size'])+'_t'+str(threshold)
                    outdict['size'] = len(cluster_set)
                    outlist.append(outdict)
        return pd.DataFrame(outlist)


    def get_next_clusters(clustersdf, min_likelihood=0):
        """Returns a new clustersdf for the subsequent day
        and a transition matrix between the input and output clustersdf.

        min_likelihood sets a lower bar on the chance that a next-day cluster is the
        same as the previous-day cluster"""

        outlist=[]
        transitions = pd.DataFrame()
        for i, row in clustersdf.iterrows():
            current_date = row['date']
            next_date = dates[dates.index(current_date)+1]

            tr_file = '%s/th_%02i/named%i_communities_transition.csv'%(current_date,
                                                                       row['threshold'],
                                                                       row['k-clique'])
            tr_matrix = pd.read_csv(tr_file, index_col=0)

            shared_elements = tr_matrix.loc[int(row['name'])]
            candidate_names = shared_elements[shared_elements>0].index

            next_clusters_filename = '%s/th_%02i/named%i_communities.txt'%(next_date,
                                                                           row['threshold'],
                                                                           row['k-clique'])
            next_clusters = read_named_cluster_file(next_clusters_filename)
```

```
            for name in candidate_names:
                outdict = {'date':next_date,
                           'threshold':row['threshold'],
                           'elements':next_clusters[int(name)],
                           'k-clique':row['k-clique'],
                           'name':name,
                           'size':len(next_clusters[int(name)]),
                           'id':(str(next_date)+'_k'+str(row['k-clique'])+
                                 '_t'+str(row['threshold'])+'_i'+str(name))}

                total_elements = set(next_clusters[int(name)]) | set(row['elements'])
                likelihood = 1.0*shared_elements[name]/len(total_elements) #normalizing here..
                if likelihood > min_likelihood:
                    outlist.append(outdict)
                    transitions.loc[row['id'], outdict['id']] = likelihood

        return pd.DataFrame(outlist).drop_duplicates('id'), transitions.fillna(0)


    def cluster_post_volume(cluster):
        """ Returns the volume of posts that contribute to the cluster,
        by combination. This is a dataframe of


        You can then take the max, min, mean, etc."""

        weighted_edgelist_file = '%s/weighted_edges_%s.txt'%(str(cluster.loc['date']),str(clus
        df = pd.read_csv(weighted_edgelist_file, sep=' ', header=None, names=['Tag1', 'Tag2',

        collect = []
        for a, b in itertools.combinations(list(cluster.loc['elements']), 2):
            count =  df[((df['Tag1']==a) & (df['Tag2']==b))|((df['Tag1']==b) & (df['Tag2']==a)
            collect.append({'Tag1':a, 'Tag2':b, 'count':count})

        return pd.DataFrame(collect)
```

We define a class object to aggregate the information needed to generate a plot of the cluster:

```
    class cluster_drawing(object):
        text_properties = {'size':12,
                           'fontname':'sans-serif',
                           'horizontalalignment':'center'}
    def __init__(self, contains, uid=None):
        if isinstance(contains, (set,frozenset)): #convenience conversion of set to list.
            contains = list(contains)

        if isinstance(contains, list):
            self.is_leaf = False
            self.contents = []
            for element in contains:
                if isinstance(element, basestring):
                    self.contents.append(cluster_drawing(element))
                else:
                    self.contents.append(element)
            self.linewidth = 1
        elif isinstance(contains, basestring):
            self.is_leaf = True
            #self.text = contains.encode('ascii', 'ignore')
            self.text = contains
            self.text = contains.decode('utf-8', 'ignore')
            self.linewidth = 0

        self.bottom = 0
        self.center = 0
        self.pts_buffer = 4
        self.uid = uid

    def get_list(self):
        if self.is_leaf:
            return [self.text]
        else:
            return [item for x in self.contents for item in x.get_list()] #flat list

    def get_set(self):
        return set(self.get_list())

    def get_by_name(self, name):
        if self.is_leaf: return None

        if self.uid == name:
            return self
        else:
            for x in self.contents:
                obj = x.get_by_name(name)
                if obj == None:
                    continue
                else:
                    return obj
            return None

        def get_uids(self):
            if self.is_leaf:
                return []
            else:
                uid_list = [item for x in self.contents for item in x.get_uids()] #flat list
                if self.uid != None:
                    uid_list.append(self.uid)
                return uid_list

        def score(self):
            """Get the score for the full (recursive) contents"""
            score=0
            this_list = self.get_list()
            for word in set(this_list):
                indices = [i for i, x in enumerate(this_list) if x == word]
                if len(indices)>1:
                    score += sum([abs(a-b) for a, b in itertools.combinations(indices, 2)])
            return score

        def order(self, scorefunc):
            """Put the contents in an order that minimizes the score of the whole list"""
            if not self.is_leaf:
                best_score = 10000000
                best_order = self.contents
                for permutation in itertools.permutations(self.contents):
                    self.contents = permutation
                    new_score = scorefunc()
                    if new_score < best_score:
                        best_score = new_score
                        best_order = permutation
                self.contents = best_order

                [element.order(scorefunc) for element in self.contents]


        def set_height(self, ax):
            if self.is_leaf:
                #have to mockup the actual image to get the width
                self.image_text = ax.text(0, 0, self.text, **self.text_properties)
                plt.draw()
                extent = self.image_text.get_window_extent()
            self.height = extent.y1 - extent.y0
        else:
            self.height = (sum([x.set_height(ax) for x in self.contents]) +
                           (len(self.contents)+1)*self.pts_buffer)
        return self.height

    def set_width(self, ax):
        if self.is_leaf:
            #have to mockup the actual image to get the width
            self.image_text = ax.text(0, 0, self.text,
                                           transform=None, **self.text_properties)
            plt.draw()
            extent = self.image_text.get_window_extent()
            self.width = extent.x1 - extent.x0 + self.pts_buffer
        else:
            self.width = max([x.set_width(ax) for x in self.contents]) + 2*self.pts_buffer
        return self.width

    def set_center(self, x):
        if not self.is_leaf:
            [child.set_center(x) for child in self.contents]
        self.center = x


    def set_bottom(self, bottom=0):
        """Sets the bottom of the box.
        recursively sets the bottoms of the contents appropriately"""
        self.bottom = bottom + self.pts_buffer

        if not self.is_leaf:
            cum_height = self.bottom
            for element in self.contents:
                element.set_bottom(cum_height)
                cum_height += element.height + self.pts_buffer

    def layout(self, ax):
        if not self.is_leaf:
            [child.layout(ax) for child in self.contents]

        plt.box('off')
        self.set_width(ax)
        self.set_height(ax)
        ax.clear()

    def draw(self,ax):
        if not hasattr(self, 'width'):
            print 'Must run `layout` method before drawing, preferably with dummy axis'

        if self.is_leaf:
            self.image_text = ax.text(self.center, self.bottom, self.text,
                                      transform=None, **self.text_properties)
        else:
            [child.draw(ax) for child in self.contents]
            ax.add_patch(plt.Rectangle((self.center-.5*self.width,self.bottom),
                                         self.width, self.height,
                                         alpha=.1, transform=None))
        ax.set_axis_off()
```

We define several functions which intermediate between the visualization object and the clusters as they have been imported:

```
def make_elements(clustersdf, k_min=0, k_max=25, order=False):

    prev_elements =[]
    for k, k_group in clustersdf.groupby('k-clique', sort=False):
        if k<k_min: continue
        if k>k_max: continue
        elements = []
        for i, row in k_group.iterrows():
            cluster_elements = row['elements']
            cluster_list = [] #this is what we will eventually pass to the class initializ
            for prev_element in prev_elements:
                prev_set = prev_element.get_set()
                if prev_set <= cluster_elements: #set 'contains'
                    cluster_elements = cluster_elements - prev_set
                    cluster_list = cluster_list + [prev_element]

            cluster_list = cluster_list + list(cluster_elements)
            elements.append(cluster_drawing(cluster_list, row['id']))

        prev_elements = elements

    a = cluster_drawing(elements)
    if order:
        a.order(a.score)
    return a
def draw_transition(a, b, tr_matrix, ax):
    for a_id in a.get_uids():
        for b_id in b.get_uids():
            try:
                likelihood = tr_matrix.loc[a_id, b_id]
            except KeyError: # if either don't show up in the transition matrix, they don'
                continue
            if likelihood > 0:
                #print a_id, b_id, likelihood
                a_object = a.get_by_name(a_id)
                b_object = b.get_by_name(b_id)

                ax.plot([a_object.center+.5*a_object.width, b_object.center-.5*b_object.wi
                         [a_object.bottom, b_object.bottom],
                         color='b', alpha=likelihood**2, transform=None)

                ax.plot([a_object.center+.5*a_object.width, b_object.center-.5*b_object.wi
                         [a_object.bottom+a_object.height, b_object.bottom+b_object.height
                         color='b', alpha=likelihood**2, transform=None)

    ax.set_axis_off()
```

Finally, we are ready to make a visualization:

```
fig = plt.figure(figsize=(18,23))
ax = plt.gca()
ax_test = ax.twinx()

prev_elements = None
transition = None
k_min=4

current_df = cu.get_clusters_with_keyword(date='20150618', threshold=5, keyword='charlesto

for i in range(2):

    center = 200*i+200
    bottom = 120
    current_elements = cu.make_elements(current_df, k_min=k_min)
    current_elements.layout(ax_test)
    current_elements.set_bottom(bottom)
    current_elements.set_center(center)
    current_elements.draw(ax)

    if prev_elements != None:
        print i
        cu.draw_transition(prev_elements, current_elements, transition, ax)

    prev_elements = current_elements
    current_df, transition = cu.get_next_clusters(current_df, min_likelihood=.2)
    datestr = dateutil.parser.parse(current_df['date'].iloc[0]).strftime('%B %d %Y')
    ax.text(center, bottom, datestr, va='top', ha='center', transform=None, fontsize=14)

ax.set_axis_off()
ax_test.set_axis_off()
```

Appendix F: Comparison of hashtags-only analysis with full message analysis

In this paper we explored clusters formed using only the hashtags present in messages. Doing so reduces the total number of messages that contribute to the network (as less than 10 percent of messages contain the requisite 2+ hashtags) and the total number of connections made by each message. The smaller volume of messages eases the computational burden to a level that should be manageable for those wishing to replicate this analysis on a modern desktop computer. To further speed computation, consider increasing the threshold for the number of messages which must be present for an edge to be represented in the semantic network.

As hashtags are a subset of all words found in a message, clusters built on hashtag cooccurrences contain a subset of the words that would be found in the same weight and k-clique cluster constructed from the full message. In order to use complete messages, the analyst should exclude stopwords, special characters, urls, and platform specific markers. The analyst may choose to ‘stem’ each word, or to find the root of each word such that, for instance, ‘shoot’, ‘shoots’, and ‘shooting’ all count toward the same total.

To give a sense for how results vary as the type of words included changes, the table below describes statistics for two cases that a user may consider. In the first case, all words within a message are used to create edges in the semantic network (excluding stopwords, URLs, and twitter-specific notation), with an inclusion threshold of 20. The second case looks at Hashtags alone, with a threshold of 2. These present similar computational burdens, and thus make for a pragmatic comparison.

Table 1: Statistics for June 16, 2015

Measure	All Words (th: 20)	Hashtags Only (th: 2)	Ratio
Messages Used	1,324,115	104,931	12.7
Total Words	10,893	16,297	0.67
Total Edges	143,043	43,645	3.3
Mean Degree	13.13	2.67	4.9
Mean Edge Weight	62.9	6.0	10.5
Cliques	34,348	19,039	1.8
Clusters	548	3,945	0.14
Median k	7	4	NA
Max k	37	17	NA
Mean Cluster Size	33.7	9.9	3.4

The all-words analysis uses all (english language) messages in the dataset, approximately 13 times as many as the hashtags-only analysis. Because of the filter, however, the total number of words present in its semantic network is less than that of the hashtags-only set. This speaks to the long-tail of low-frequency connections made in the dataset that are more aggressively pruned by the higher inclusion threshold.

As the number of connections suggested by a message scales as the square of the number of words it contains, the mean degree of the all-words network is approximately five times that of the hashtags-only network. Due to both the higher absolute number of edges and stronger pruning, the mean number of times a connection appears in the dataset is an order of magnitude larger in the all-words case. The higher connectivity of the all-words network implies a smaller number of larger, more tightly connected clusters.