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

No author given

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 highlights how world events become connected to existing conversations. This approach has applicability to the study of framing processes, and may reveal how contentious groups connect evolving events to their larger narratives.

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

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

by Huberty (2013) and Tumasjan, Sprenger, Sandner, and Welpe (2010); or processes such as social movements, as demonstrated by Agarwal, Bennett, Johnson, and Walker (2014) and 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.

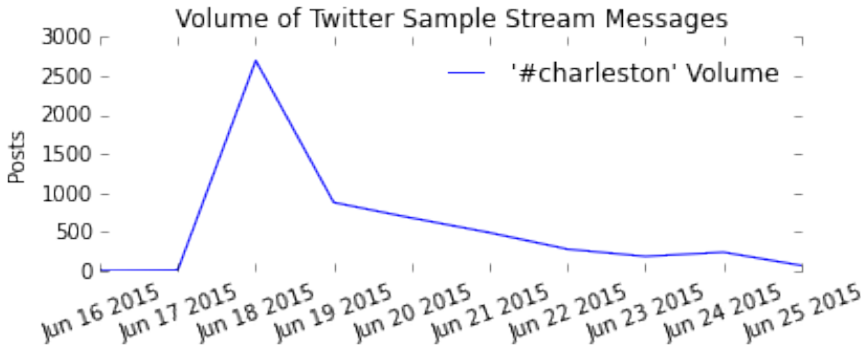

Figure 1. Standard methods of social media analysis include keyword volume tracking (shown here), sentiment analysis, and supervised categorization.

For example, if we are interested in understanding the national conversation in reaction to the shooting at Emanuel AME church in Charleston, South Carolina in June of 2015, we could plot a timeseries of the volume of tweets containing the hashtag `#charleston`, as seen in Figure 1. This tells us something about how many people are engaging with the topic, but little about the ways they are interpreting the event or how they connect the event to their preexisting ideas about gun violence.

Lee, Cesare, Shojaie, and Spiro (2015).

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, Naaman, and Gravano (2011), Ritter, Cherry, and Dolan (2010), and Zubiaga, Spina, Fresno, and Martínez (2011). Such methods can give aggregated insight into the sentiment expressed, but not into the ways the events are being framed within existing conversations.

Because they look at individual messages and not at the relationships between ideas within messages, these techniques are unable to infer from the data coherent patterns of thought that signify interpretation of the events' deeper meanings. Interpretation depends upon making connections between the events as they happen and other concepts in the public discourse. One way to measure these connections is to look at a network of co-citations surrounding our topic of interest, as has been demonstrated by Cogan and Andrews (2012) and Smith and Rainie (2014). This technique represents hashtags as nodes on a network, and each message containing two or more hashtags contributes to the weight of an edge between these nodes. This type of analysis is helpful in that it helps us begin to understand the structure of the discourse. If we perform k-clique clustering on the network, we can see which sets of connections form coherent and conversations, as seen in Figure 2.

In this example there seem to be two distinct conversations happening with regards to the event: the first a description of the shooting itself and

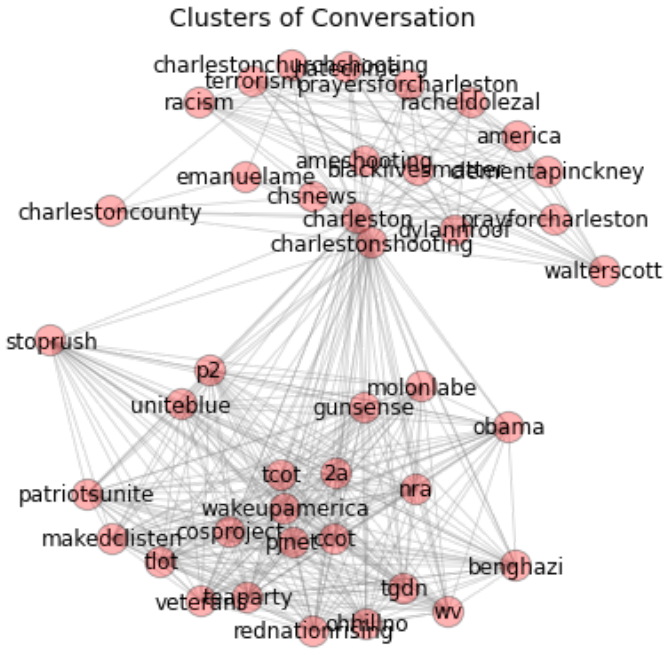

Figure 2. K-clique clustering reveals coherent structures in the hashtag co-citation network that can serve as proxies for conversations in the larger national discourse.

the human elements of the tragedy. A second conversation focuses on the larger national-scale political conflicts to which the event points. While each of the conversations is motivated by the same event, they are distinct from one another in the language they use and in the connections they draw.

1.1 The contributions of this paper

We may hypothesize that how these conversations develop over time will influence the social and political response to the event. In this paper we will explore methods of identifying and tracking the development of these conversations over time. We show how these conversation clusters can be identified

and visualized, exploring how certain elements form the core of a conversation, while other elements circulate at the periphery. We then demonstrate a method for tracking the evolution of these conversational clusters over time, by relating clusters identified on one day with clusters emerging on subsequent days. This allows us to qualitatively see what new concepts are being included into the discussion, and quantitatively track engagement in these conversations, as distinct from mere references to keywords.

Due to the computation-intensive nature of this analysis, we chose to implement the data manipulation algorithms in both Python and Unicage shell scripts,[2] 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 image in Figure 2 is based upon network closeness between hashtags. 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).

The clusters themselves are then defined by k-clique community detection algorithms implemented in the COS Parallel library, developed by Gregori, Lenzini, and Mainardi (2013) and their use is demonstrated in the appen-

[2]For a description of Unicage development tools, see Tounaka (2013)

dices.

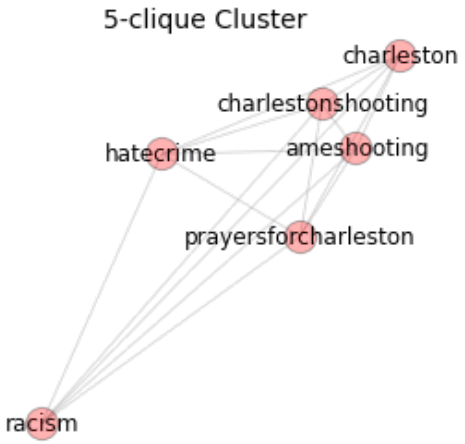

Figure 3. Clusters with higher `k` *value are smaller and more tightly connected, representing a more coherent or focused conversation.*

Every node in a cluster must be able to participate in at least one fully connected subgroup of size `k` with the other members of the group. Thus, the metric `k` determines how strict we are about closeness between keywords when defining the boundaries of a particular cluster. 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.

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.

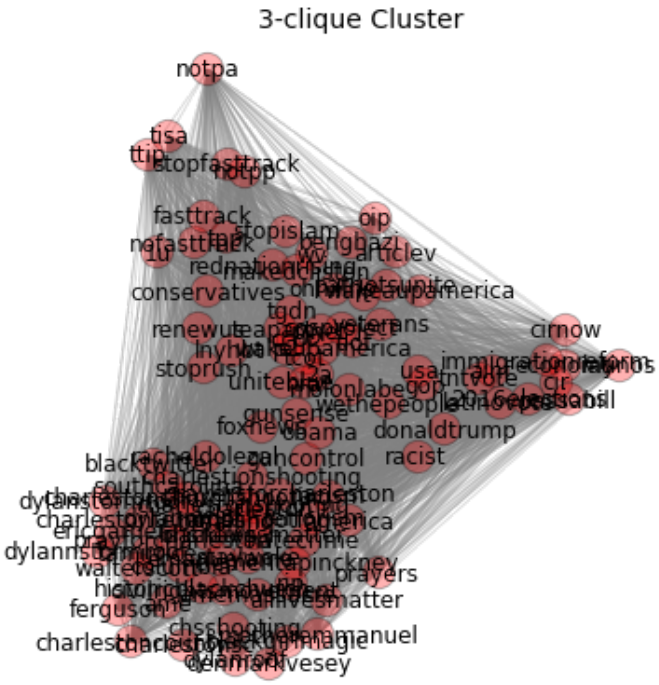

Figure 4. Clusters with lower `k` *value are larger and less less tightly connected, representing more diffuse conversation. They may have smaller clusters of conversation within them.*

3. REPRESENTING CONVERSATIONAL CLUSTERS AS NESTED SETS

Tight conversational clusters (high `k`) must necessarily be contained within larger clusters with less stringent connection requirements (low `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 favor of something which allows us

to observe the nested relationships the conversations have with one another.

In this representation, we are able to observe the tightly clustered 5-clique conversation in context of the 4-clique conversation it inhabits, and the neighboring 4-clique conversations that together inhabit the larger discourse.

4. TRACKING CONVERSATIONS CHRONOLOGICALLY

In order to track how elements of conversation weave into and out of the general discourse, 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-citations 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 the conversational clusters shown in Figure 5 fall in volume over the 10 days subsequent to the initial event, paralleling the decay in pure keyword volume seen in Figure 1.

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

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

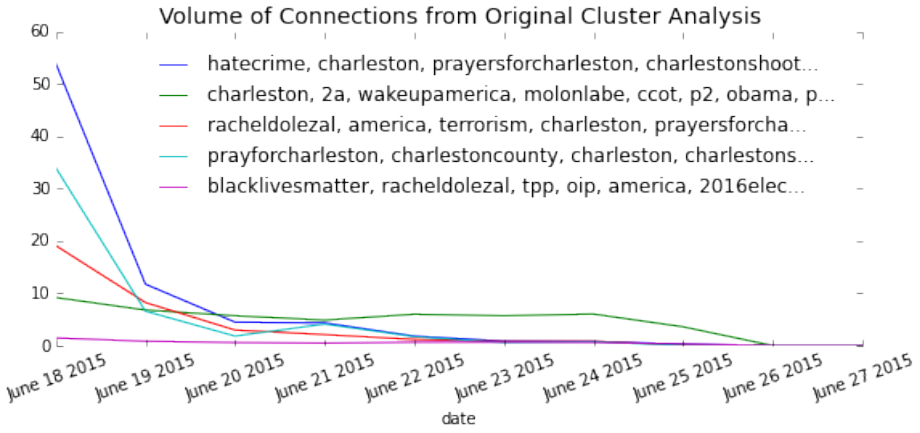

Figure 6. Tracking the volume of connections made in a single day's clusters (e.g. co-citations) reveals how the specific analogies made immediately after the event maintain their relevance.

immediately following time period.

Palla, Barabási, and Vicsek (2007) discuss how communities of individuals develop over time and change. We can use the same techniques to track continuity of conversational clusters. The most basic way to do this is to count the fraction of elements of a conversational cluster at time 1 that are present in each conversational cluster at time 2, 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, as seen in Table 1.

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

Table 1. A transition matrix contains the likelihood that a cluster at one timeperiod (rows) corresponds to a cluster in the subsequent timeperiod (columns).

Cluster ID	t2-cl1	t2-cl2	t2-cl3	...
t1-cl1	0.68	0.2	0.0	...
t1-cl2	0.0	0.85	0.03	...
t1-cl3	0.13	0.04	0.45	...
...	...	...	...	...

from the networks at time 1 and time 2. This reduces the number of possible pairings between days, yielding more specificity in our intra-day transition matrices.

These transition matrices can be used to infer how clusters present in the first day's analysis correspond to clusters in the second day, and so forth. These are visualized as a set of nested cluster diagrams, with traces linking likely clusters together, as seen in Figure 7. Heavier traces between clusters imply more confidence in the transition between the two sets.

The volume of messages forming the edges of each cluster are shown plotted day by day in Figure 8. As the linkage between subsequent clusters is probabilistic, in this plot so to are the links connecting volume measures, with the weight of each link proportional to its likelihood. In the present example, we can see how following expressions of the event itself, the conversation evolves into a discussion of the relationship between the violence and the ongoing issue of racism and its connection with the confederate flag. Parallel to and as an undercurrent of these conversations is the ongoing discussion of

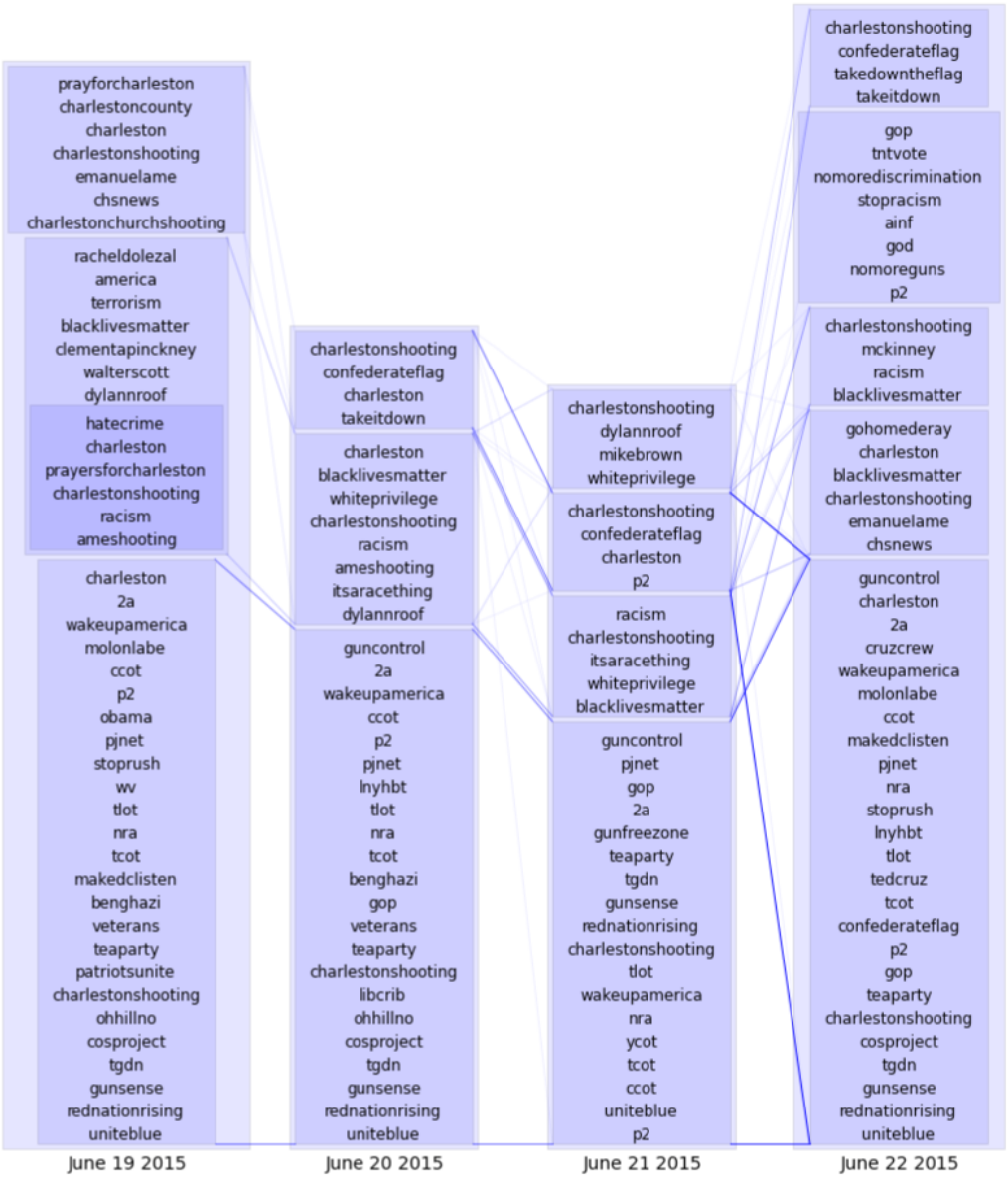

Figure 7. Weighted traces connect conversation clusters for four days following the shooting. Conversations change over time, as certain components fall out of the cluster, and other components are added.

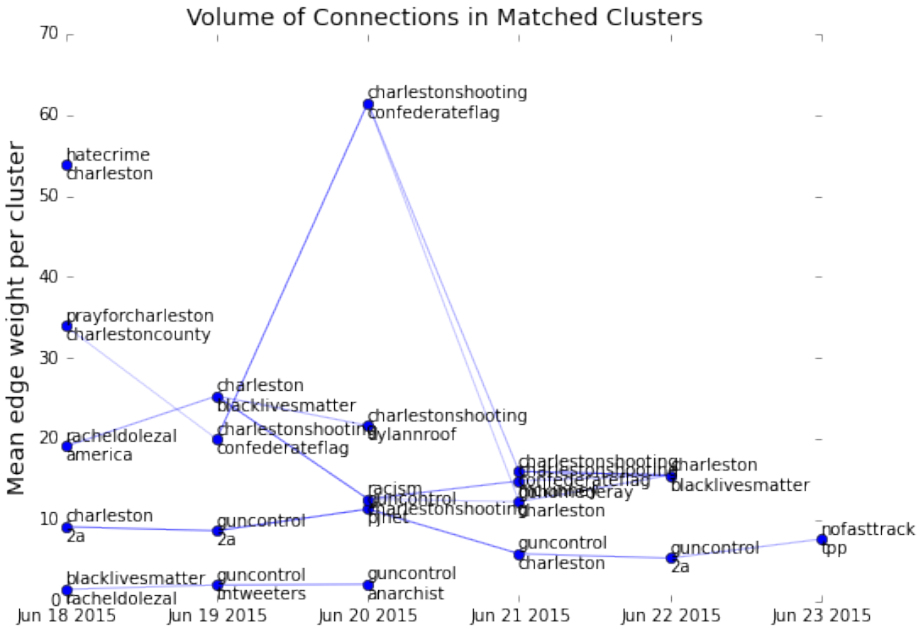

Figure 8. Plotting over time the mean connection volume in in each conversational cluster, showing links according to the probabilistic transition matrix. Clusters show relative consistency, with new terms occasionally raising participation.

gun control.

5. CONCLUSION

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 hashtag co-citations 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 connection graph 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

The full set of scripts, and associated documentation, can be found at `www.github.com\ removed to anonymize`.

REFERENCES

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, C. A. (2015), "Taming Big Data: Using App Technology to Study Organizational Behavior on Social Media," *Sociological Methods & Research*, 0049124115587825–.

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

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*.

DiGrazia, J. (2015), "Using Internet Search Data to Produce State-level Measures: The Case of Tea Party Mobilization," *Sociological Methods & Research*, 0049124115610348–.

Gregori, E., Lenzini, L., and Mainardi, S. (2013), "Parallel k-clique commu-

nity detection on large-scale networks," *IEEE Transactions on Parallel and Distributed Systems*, 24(8).

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

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

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*, 0049124115605339–.

Palla, G., Barabási, A., and Vicsek, T. (2007), "Quantifying social group evolution," *Nature*.

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*, pp. 172–180.

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

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

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

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*, 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:

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

```
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
```

```
#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 متحره
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...
```

```
# 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']))
```

```
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.tx
                                                                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.cs
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: listwordpairings.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

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                                                                |
    jq -c '{time: .timestamp_ms, hashtag: [.entities.hashtags[]?.text]}' |
    grep "time"                                                                  |
    grep "hashtag"                                                               |
    grep -v ':null'                                                              |
    tr -d '{}[] '                                                                |
    tr ':' ','                                                                   |
    fromcsv                                                                      |
    # 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:count

    # run 5 processes in parallel
    touch ${semd}/sem.$n
   } &
   if [ $((n % 5)) -eq 0 ]; then
     eval semwait ${semd}/sem.{$((n-4))..$n}
     eval rm ${semd}/sem.*
   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                                                          |
    jq -c '{time: .timestamp_ms, hashtag: [.entities.hashtags[]?.text]}' |
    grep "time"                                                             |
    grep "hashtag"                                                          |
    grep -v ':null'                                                         |
    tr -d '{}[] '                                                           |
    tr ':' ','                                                              |
    fromcsv                                                                 |
    # 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:count

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

#semwait "${semd}/sem.*"
wait
eval rm ${semd}/sem.*

exit 0
```

Step 2: wgtededgegen.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: unwgtededgegen.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: backtoorg_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: computetransitionlikelihoods.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


for curr_date in $(cat $tmp-date-list); do

  next_date=$(mdate ${curr_date}/+1)

  n=0
  echo ${workd}/${curr_date}/th_*/named*_communities.txt    |
  tarr                        |
  ugrep -v '\*'                   |
  while read curr_filename; do
    n=$((n+1))
    echo ${curr_filename} $n

    {
      # extract end of filepath (ex: th_02/named3_communities.txt)
      tmp_next_filename=$(echo ${curr_filename} | sed -e 's/\// /g' | self NF-1/NF | sed -e 's/

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

      # create next_date's filepath
      next_filename=${workd}/${next_date}/${tmp_next_filename}

      if [ ! -s ${next_filename} ]; then
```

```
  touch ${semd}/sem.$n
  continue
fi

# create sets of tag for each community
tarr num=1 ${curr_filename}       |
# 1:community id 2:tag
# exclude dupulicated tag in each community
msort key=1/2             |
uniq                  |
yarr -d, num=1                > $tmp-curr_cluster.$n
# 1:community id 2:tagset(csv)
#  ex) 0 tag1,tag2,tag3,...
#      1 tag1,tag3,...
[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

# create sets of tag for each community
#   same as ${curr_filename}
tarr num=1 ${next_filename}       |
msort key=1/2             |
uniq                  |
yarr -d, num=1                > $tmp-next_cluster.$n
[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

joinx $tmp-curr_cluster.$n $tmp-next_cluster.$n   > $tmp-joinx_cluster.$n
# 1:id(curr) 2:tagset(curr) 3:id(next) 4:tagset(next)

self 1 3 2 4 $tmp-joinx_cluster.$n    > $tmp-joinx_cluster_wk.$n
# 1:id(curr) 2:id(next) 3:tagset(curr) 4:tagset(next)

${shelld}/intersection.test $tmp-joinx_cluster_wk.$n > $tmp-likelihood.$n
# 1:id(curr) 2:id(next) 3:count

# create map: index=id(curr) columns=id(next)
map num=1 $tmp-likelihood.$n      > $tmp-likelihood.map.$n

# csv file name
dirname=$(dirname ${curr_filename})
mkdir -p $dirname
csv_filename=$(basename ${curr_filename} '.txt' | gawk '{ print "'${dirname}'/"$0"_trans
[ $(plus $(echo "${PIPESTATUS[@]}")) -eq "0" ] || ERROR_EXIT

# comvert to csv
tocsv $tmp-likelihood.map.$n          > ${csv_filename}
```

```
      # run 5 processes in parallel
      touch ${semd}/sem.$n
    } &

   if [ $((n % 5)) -eq 0 ]; then
     eval semwait ${semd}/sem.{$((n-4))..$n}
     eval rm ${semd}/sem.*
   fi

  done

  #semwait "${semd}/sem.*"
  eval rm ${semd}/sem.*

done


# delete tmp files
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	08h:27m:43s
2. Weighted Edgelists	00h:00m:10s	00h:03m:32s
3. Unweighted Edgelists	00h:13m:36s	00h:08m:11s
4. Maximal Clique Identification	00h:10m:13s	00h:02m:18s
5. k-Clique Percolation	00h:05m:45s	00h:08m:53s
6. Mapping back to Text	00h:19m:56s	00h:09m:56s
7. Computing transition likelihoods	00h:03m:15s	05h:47m:51s
Total time	**39h:27m:28s**	**14h:48m:24s**

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'), c
        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(clust
        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't
                    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()
```