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='charleston

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()
```