Capstone: Displaying Large Organizational Networks – Technical Guide

The purpose of this document is to walk through the data acquisition and processing steps that were required to clean and prepare the data used in the app and the process that the app uses to generate network maps. Please note that terminology is not defined in this document. Definitions are in either the app or the User Guide and Final Report.

Raw Data and Data Pre-Processing

To review this process in detail, please refer to the project's Data Wrangling Report[1] for a detailed explanation and data wrangling IPyNotebook[2] for full code details.

As indicated in the User Guide, the data for the app was collected using an online survey. The survey software, LimeSurvey, as provided by the University of Vermont (UVM), provided raw survey returns as a comma-separated spreadsheet (.csv), in which each row represents the response from one survey respondent. With all of the survey questions provided together in a single output, it was necessary to extract the network from the overall survey returns. Obtaining the survey returns took two primary steps: changing the organizational names in the survey responses and cutting non-network data.

Anonymizing Organizational Names

The raw survey returns included a set of organizational names designed to be recognizable by the survey respondents and made assumptions about colloquial usage and the survey context. That is, some organizations were referred to by names that would be recognized by staffers of that organization while completing the survey but would not be recognizable outside that context and might not be different from other organizational labels and acronyms. This was necessary to limit the burden on respondents but created a need to recast these names for analysis. Further, once the data was made ready for public release, the organizational names needed to be anonymized to meet requirements of the UVM Institutional Review Board (IRB) or protecting Personally Identifiable Information (PII).

To complete this, two Python dictionaries were created that matched the organizational names in the survey to, first, a unique organizational acronym, and then matched the acronym to an anonymous identifier, as described in the dataset's codebook. This two-step process was used to match with existing transformation keys used in R; preparation for academic research did not, initially, include the anonymization step, as this was not necessary until the data were to be publicly released. The follow code implemented these transformations. Numeric indices were

[1] On Github: https://github.com/wmirecon/Visualizing_Gov_Nets/blob/master/data%20wrangling%20report.pdf
[2] This notebook recreated a process previously completed using R. IPyNB on GitHub: https://github.com/wmirecon/Visualizing_Gov_Nets/blob/master/ipython%20script%20prep%2014aug17.ipynb
R script version: https://github.com/wmirecon/Visualizing_Gov_Nets/blob/master/data%20prep%20r%20code.R

used to preserve row and column labels as part of the dataset; preserving this structure aids in processing in network data.

```
# convert names to internal acronyms
name_key = pd.read_csv(r'C:…\name key.csv')
name_key = dict(zip(name_key.org_name, name_key.int_acronym))

# convert external acronyms to internal acronyms
id_key = pd.read_csv(r'C:…\acronym key.csv')
id_key = dict(zip(id_key.ext_id, id_key.int_id))

# convert internal acronyms to anonymous ids
anon_key = pd.read_csv(r'C:…\anonymization key.csv')
anon_key = dict(zip(anon_key.int_id, anon_key.anon_id))

# assign anonymous id's and check for failures
prep_data['anon_id'] = prep_data.int_id.map(anon_key)
prep_data.loc[prep_data.int_id.isnull()]

# replace existing ID's with anonymous ids
prep_data[0] = prep_data.anon_id
prep_data.set_value(0,0,'anon_id')
prep_data.drop([1,2,3], axis = 1, inplace = True)
prep_data.drop('int_id', axis = 1, inplace = True)
prep_data.drop('anon_id', axis = 1, inplace = True)
prep_data.head(10)
```

Section Separation

With the names adjusted, the data could be sectioned into organizational network links and the subnetwork filters that the app applies for created network slices. The data for filters, identified as bimodal data sections in the following code, were then saved, and later merged with the organization list. This list was used by the app for filtering.

```
# separate out bimodal data sections
# identify last column of bi-modal data sets; found at:
print(prep_data.iloc[:,[63,64,65]].head())
bimodal_sets = prep_data.drop(list(range(68,prep_data.shape[1] + 3)),
axis = 1)
bimodal_sets.head(10)

# limit data to just network data sets, fill NaN's with 0's, transpose,
and change to anon_ids
combined_subnets = prep_data.drop(list(range(4,68)), axis = 1)
combined_subnets.fillna(0, inplace = True)
combined_subnets_t = combined_subnets.transpose()

# the data is transposed so that a new column may be made that adds in
data on subnet labels and for assigning anon_ids
# this is typically easier to do in columns than rows. transposing
allows for doing these operations as columns
# final products will be transposed back to produce properly-configured
NSQ's
```

```
# replace first new first column with internal ids and check for errors
combined_subnets_t['int_id'] =
                  combined_subnets_t.iloc[:,0].map(name_key)
combined_subnets_t.loc[combined_subnets_t['int_id'].isnull()]

# replace first new first column with anonymous ids and check for
errors
combined_subnets_t['anon_id'] = combined_subnets_t.int_id.map(anon_key)
combined_subnets_t.loc[combined_subnets_t['anon_id'].isnull()]

# replace anon_ids as the first column (necessary for network matrix
structure) and remove trailing ids columns
combined_subnets_t[0] = combined_subnets_t.anon_id
combined_subnets_t.set_value(0,0,'anon_id')
combined_subnets_t.drop('int_id', axis = 1, inplace = True)
combined_subnets_t.drop('anon_id', axis = 1, inplace = True)
```

Separate Network Matrices

Our survey collected five different functional subnetworks, identified in the app. Within the raw survey returns, the data for each subnetwork were mixed together. Rather than separating out link types, the raw .csv dataset sorted data by each potential interaction partner, so that the data indicated if an organization interacted with any one organization, and whether it did so for each of the five interaction types that form the functional subnetworks, before moving on to the next organization. The next step was to separate these different matrices and obtain five separate, functional subnetwork-specific matrices. This was done by transposing the data, to allow for easier filtering by row rather than column, and labeling each row with a functional subnetwork label. Once this was done, the data were filtered by these labels and re-transposed.

```
subnets = ['is','ta','rt','fs','pc']
# create a list that repeats the subnet list in the correct order and
is equal in length to the number of rows in the dataset
subnet_list = ['subnet']
iterations = int( ( combined_subnets_t.shape[0] - 1 ) / 5 )
subnet_iterations = subnets * iterations

for item in subnet_iterations:
    subnet_list.append( item )

# append that subnet list to the dataset for filtering
combined_subnets_t['subnet'] = subnet_list

# filter data and obtain NSQs for each subnetwork
# filter
is_t = combined_subnets_t.loc[
                  combined_subnets_t['subnet'].isin(['subnet','is'])]
ta_t = combined_subnets_t.loc[
                  combined_subnets_t['subnet'].isin(['subnet','ta'])]
pc_t = combined_subnets_t.loc[
                  combined_subnets_t['subnet'].isin(['subnet','pc'])]
rt_t = combined_subnets_t.loc[
                  combined_subnets_t['subnet'].isin(['subnet','rt'])]
fs_t = combined_subnets_t.loc[
                  combined_subnets_t['subnet'].isin(['subnet','fs'])]
```

```
# cut final column
is_t = is_t.iloc[:,list(range(is_t.shape[1] - 1))]
ta_t = ta_t.iloc[:,list(range(ta_t.shape[1] - 1))]
pc_t = pc_t.iloc[:,list(range(pc_t.shape[1] - 1))]
rt_t = rt_t.iloc[:,list(range(rt_t.shape[1] - 1))]
fs_t = fs_t.iloc[:,list(range(fs_t.shape[1] - 1))]

# re-transponse
is_nsq = is_t.transpose()
ta_nsq = ta_t.transpose()
pc_nsq = pc_t.transpose()
rt_nsq = rt_t.transpose()
fs_nsq = fs_t.transpose()
```

Transform NSQ Matrix into Network Edgelist

Once separated and re-transposed, the data are in the form of five separate non-square (NSQ) matrices. Network packages in R (or Python) are unable to handle data in this structure.[3] Instead, these datasets must be transformed into either a square matrix or a network edgelist. The edgelist structure is easier to obtain and R's network analytic packages are better able to integrate organizational data with the link data when the data are formatted as an edgelist. In the final pre-processing step, a function was defined that sorted through each cell of the NSQ and listed out the organizational names for each interacting pair, or network dyad, when an interaction was found. This function was then applied to each NSQ to generate five different edgelists. These edgelists were then used as the raw input data for the app.

```
def nsq_to_el( nsq ):
    import pandas as pd
    el = []                    # a list to store edges in

    # obtain row and column counts for use in for-loops
    n_init = nsq.shape[0]
    n_respond = nsq.shape[1]

    # loop through the cells of the dataframe that contain link data
    # links are indicated by values 1 and 2
    # if a link is found, store the initiating and responding orgs (row
and column names) in a tuple
    # and append the tuple to the list of edges
    for init in range(n_init):
        for respond in range(n_respond):
            if ( nsq.iloc[init, respond] in ('1','2') ):
                edge = ( nsq.iloc[init, 0], nsq.iloc[0,respond])
                el.append(edge)

    # convert edgelist list into a data frame
    el_df = pd.DataFrame.from_records(el, columns = ['from','to'])
    return(el_df)
```

[3] With the exception of when the rows and columns represent different sets of nodes. If the nodes on the rows and the columns are from the set of nodes, such as both being organizations, then the dataset is not formatted for processing.

App Processing and Map Rendering

The above steps prepare the raw data for use in the app. The app then turns an organization list and a series of edgelists into the maps that it displays and their associated organization list. This is achieved using twelve (12) reactive functions. Reactive functions are a tool that R's *shiny* package provides. The app saves the output from a reactive function and only re-runs the code in the function when the function's inputs have changed. This arrangement speeds up processing by ensuring that code is only re-run when it is needed. Unlike normal functions, where the code is contained in brackets ("f(args){`<code>`}"), reactive functions are identified by the combined use of parentheses and brackets, generally without arguments ("r({`<code>`})"). Further, unlike other functions, reactive functions must be defined in a *shiny* app's `server.R` file and cannot be moved to a separate file. This app's reactive functions handle four processing steps. For brevity the code for each function is not included here. Please refer to `server.R` for the code for these functions.

Edgelist selection

The app must first select the appropriate data based on user settings. The organization list is common to all data. But the appropriate edgelist is determined by user settings. There is one reactive function for this task, labeled as reactive function #1 in `server.R`.

1. `get.raw.el({})` – loads the edgelist for the selection functional subnet

Organization list filter

Limiting networks based on filtering selection is carried out by first limiting the organization list to only those organizations that should be included. This filtered list is then used to filter out network dyads that should not be shown. Five reactive functions apply a user's filtering options. Each of these provides a cutout point by which processing speed can be increased by only producing new functional outputs when a user changes the associated input.

2. `select.dm({})` – Find the list of all organizations active in a user's selected policy domain
3. `select.pt({})` – Find the list of all organizations active in a user's selected policy tool
4. `select.aa({})` – Find the list of all organizations active in a user's selected action arena
5. `filter.orgs({})` – Applies the intersection or union filter to the domain, tool, and action arena results
6. `ego.filter({})` – Further refines the list of organizations based on a user's selection for focus organizations; unlike the other selection functions, this requires some input from a filtered edgelist since it seeks neighbors of the selected focus organizations

Edgelist filtering and network construction

Once the organization list has been filtered, the app filters the edgelist by only including those dyads where both interaction partners are included in the list of selected organizations.

7. `get.net({})` – apply the edgelist selection and list filtering functions to obtain an R/*igraph* network object. R/*igraph* is a network analysis package. The network object is used as an input for the output generation functions.

Output Generation

The app generates three different outputs: a network map, a table of key organizations included in the map, and a table of network statistics for the displayed map. Three outputs are included in that table so that five total specific outputs are generated.

8. `forceNetwork({})` – generates the network map from the network object
9. `key.orgs.table({})` – generates the table of key organizations and sorts the table by each organization's degree centrality
10. `node.count({})` – obtains a count of the number of nodes in the network object
11. `link.count({})` – obtains a count of the number of links in the network object
12. `net.density({})` – obtains the density of the network contained in the network object

Notes on additional application files

There are three other files that make up the app's source code:

1. `ui.R` – constructs the app's user interface
2. `function helpers.R` – defines the non-reactive functions that the app uses, including functions for edgelist filtering, applying a user's selections for domains, tools, and action arenas, and applying a user's focus organization selections
3. `look up helpers.R` – Several R vectors and look up tables (compare Python lists and dictionaries, respectively) are used in both server.R and ui.R. This file contains those vectors and look up tables.