Introduction to network comparison

CMT224/4224: Social Computing

Case Study

Dataset: GitHub Commits

> GitHub, Inc. is a provider of Internet hosting for software development and version control using Git. It offers the distributed version control and source code management (SCM) functionality of Git, plus its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, continuous integration and wikis for every project. -- Wikipedia
>
> GitHub is of course not a social media website and casual social features are arguably not its core function. However, interaction between individuals is often a fundamental trait of its features, particularly for public/team projects. E.g., collaboration on code bases, issue tracking, wikis, etc. Therefore, a secondary benefit of the data being generated as part of repository operation is the ability to analyse aspects of collaboration behaviours.

- We will explore this by modelling the commit data in a similar way to a social network with nodes as users and in a similar way to what has been undertaken for other collaborative platforms such as Wikipedia (e.g., https://arxiv.org/abs/1904.08139). We will use commit histories of various public repositories, including some from the same organisation. Exploring the commit history data does not provide an exhaustive proxy for the individual collaborative behaviours and cultures between developers, but it does provide a basis for simple, comparable modelling that we can compare against social phenomena seen more generally.
- We will also use this as the basis to explore how networks of different sizes, structures, and content can be compared, through comparing the commit behaviours between repositories and within repositories over time.
- The data used in this notebook is a subset of a larger dataset available at: https://www.kaggle.com/dhruvildave/github-commit-messages-dataset

This exercise assumes a Python 3 ipykernel environment.

If you are not running a virtual environment, run the cell below. Then go to "Kernel" on the top menu and click "Restart Kernel". Or if you are running a virtual environment, install networkx, matplotlib, etc, and jupyter/ipykernel in the virtual environment instead.

In [1]:
```
%pip install networkx numpy
```

```
Requirement already satisfied: networkx in /opt/homebrew/anaconda3/lib/python3.12/site
-packages (3.5)
Requirement already satisfied: numpy in /opt/homebrew/anaconda3/lib/python3.12/site-pa
ckages (1.26.4)
Note: you may need to restart the kernel to use updated packages.
```

In [2]:
```
import networkx as nx
import csv
from datetime import datetime, timezone
import matplotlib.pyplot as plt
import numpy as np
import glob
import operator
```

1. Building networks from the GitHub commit data

Understanding the raw dataset

Here we don't have pre-built networks to analyse, we must instead build a suitable network or networks from the data first and then undertake the analysis. The dataset in question is a collection of 34 CSV files, each spanning 12-months of commit behaviour for a single, public GitHub repository.

- Each row in a CSV file represents an individual commit and has two columns: The first is a user id and the second is the date and time the commit. The CSV files do not have headers.
- User ids are pseudonymised from GitHub usernames and shared across the data files (repositories).
- The commits for each repository have been pre-sorted in date and time descending order.

Example commit information from the dataset:

```
csv
    115089      31/12/2020 13:35:52 +0100
    118185      27/12/2020 19:34:16 +0100
    118185      24/12/2020 12:06:01 +0100
    118118      23/12/2020 16:37:20 -0800
```

Creating a network from commit behaviour with NetworkX

The data files contains commit behaviour for a single repository, however some users commit to multiple repositories.

As a result, there are different ways a network structure could be constructed. For example, we could build a 'social network' of users with connections between users that commit to the same repository. Another example could be to build a network that embeds the order in which users commit to a repository to model the potential collaboration interplay between developers.

We first need to define a scope for the network (or networks) based on some overarching research question(s) or hypothesis/hypotheses.

This *could be*: **Do patterns in cumulative commit data expose similarities or differences in the collaborative behaviour of public software projects?**

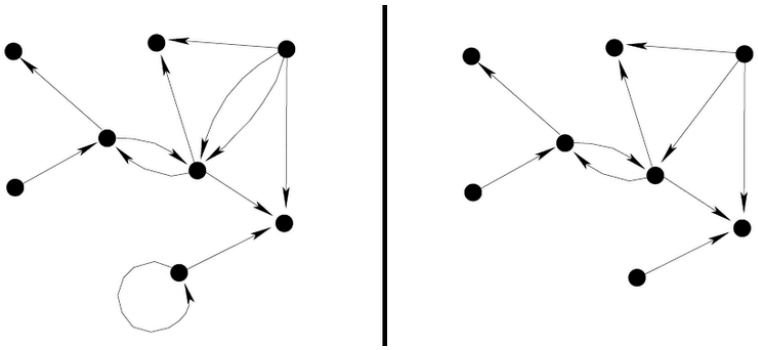
Choosing a type of Graph

A key component of the data is the temporal component of the rows (commits), whereby one commit comes after another or put another way one person commits to a repository after someone else.

Lets refer to the NetworkX documentation for a list of different types of Graphs (that it supports):

https://networkx.org/documentation/stable/reference/classes/index.html

Two immediate choices given what the edges could represent are a **Directed MultiGraph (MultiDiGraph object)** or a **Directed Graph (DiGraph object)**.



-- Lopez and Burcet (2012)

A **Directed MultiGraph** may be seen as the most appropriate choice as the data likely contains multiple instances where someone commits after someone else. Directed MultiGraphs can be seen as more complex to analyse with many 'traditional' graph theory methods available concentrating on Undirected or Directed graphs (although many have extensions continuing to be proposed in research papers). They can also offer more opportunities for a richer analysis where, for example, temporal data is embedded on the edges (see Extra: Temporal motifs).

Instead, the data could be represented as a **Directed Graph, with edge weights** that represent the number of times someone commits after someone else. This has the benefit of allowing weights to be considered for some analyses and not for others - much like the Reddit hyperlink notebook.

> **Extra**: Temporal motifs: https://snap.stanford.edu/temporal-motifs/

For this notebook, we will choose the later and focus on examining and comparing commit behaviour across and within repositories. Extending this with a Directed MultiGraph would be a good considering for future work as part of your independent study.

In [3]:
```
commit_file = "data/jupyterlab+jupyterlab.csv" # For simplicity initially, we will on

G = nx.DiGraph() # create an empty NetworkX DiGraph object
```

To help contextualise what we wish to achieve, lets consider the following simplified example. Assume that a repository only has 4 commits in total sorted in time descending order as below:

```
csv
     (user) (date and time of the commit)
row 1   3       31/12/2020 11:37:52 +0100
row 2   9       27/12/2020 14:34:23 +0100
row 3   17      23/12/2020 16:37:19 -0800
row 4   3       23/12/2020 12:11:04 -0800
```

Here we loop through the data to create a directed network by looping over each row and creating an edge (or increasing the weight) from the user in row n to the user in row $n+1$. We want the resulting network to have the following edges:

```
3 -> 9    i.e, user 3 committed after user 9

9 -> 17    i.e, user 9 committed after user 17

17 -> 3    i.e, user 17 committed after user 3
```

Alternatively, the network model could be built to represent edges in the opposite direction, i.e., a user commits *before* another user.

In [4]:
```
# Use Python's csv package to read each line of the csv file
with open(commit_file) as csvfile:
    reader = csv.reader(csvfile)

    user_of_the_next_commit = next(reader)[0] # move the iterator along by 1, getting

    for i, commit in enumerate(reader): # for each row in the file (commit to the rep
        user_of_this_commit = commit[0] # extract out the user id, the first column

        if G.has_edge(user_of_the_next_commit, user_of_this_commit):
            G[user_of_the_next_commit][user_of_this_commit]["weight"] += 1
        else:
            G.add_edge(user_of_the_next_commit, user_of_this_commit, weight = 1)

        # importantly, the user_of_this_commit being processed should now replace the
        # 'next' here refers to 'next' in a temporal sense as the file is sorted by t
        user_of_the_next_commit = user_of_this_commit
```

Or put another way. Firstly, process the first row by just extracting out the user:

user_of_the_next_commit = ***3***

Then start looping over each of the other commits (i.e., starting from the second row):

1. user_of_this_commit = **9**
2. create an edge from ***3*** to **9**
3. user_of_the_next_commit now = **9**

Now consider the third row:

1. user_of_this_commit = *17*
2. create an edge from **9** to *17*
3. user_of_the_next_commit now = *17*

Now consider the fourth row:

1. user_of_this_commit = ***3***
2. create an edge from *17* to ***3***
3. user_of_the_next_commit now = ***3***

No more rows.

Lets examine some structural properties of the created network. Refer back to the notebooks in previous weeks for more information on each measure.

In [5]:
```
print(f"Number of nodes: {G.number_of_nodes()}")
print(f"Number of edges: {G.number_of_edges()}")
print(f"Total of edge weights: {G.size(weight='weight')} (Number of commits - 1) \n")

print(f"Number of strongly connected components: {nx.number_strongly_connected_compon
print(f"Number of weakly connected components: {nx.number_weakly_connected_components
```

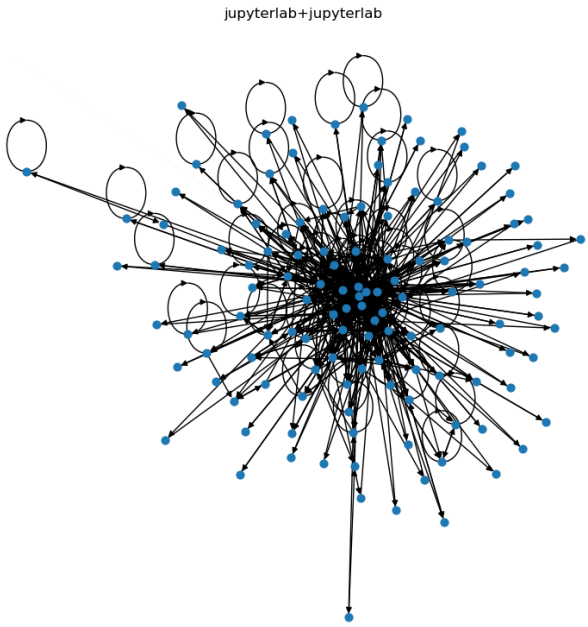
```
Number of nodes: 126
Number of edges: 677
Total of edge weights: 3540.0 (Number of commits - 1)

Number of strongly connected components: 1
Number of weakly connected components: 1
```

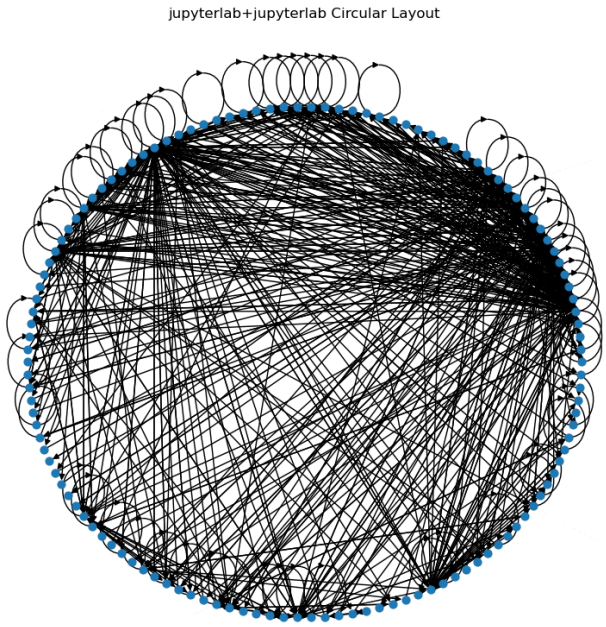
Some immediate observations:

- The number of edges and the total weight on the edges is relatively small in comparison to the other notebooks seen so far (e.g. Reddit hyperlinks);
- The network is unsurprisingly weakly connected, but perhaps more surprisingly is also strongly connected.

As the number of nodes and edges is reasonably small, lets visualise the network:

In [6]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 8))
nx.draw(G, ax=ax1, node_size=40)
nx.draw_circular(G, ax=ax2, node_size=40)
ax1.set_title("jupyterlab+jupyterlab")
ax2.set_title("jupyterlab+jupyterlab Circular Layout")
plt.tight_layout()
plt.show()
```





Self loops?

This is the first time we have seen a case where a node has an edge going from itself to itself. This is called a 'self loop'. On reflection, this is perhaps unsurprising here as Git repositories do not assume that different people will be committing each time.

However, this creates another decision point in the modelling of the network as to whether self loops should be kept or removed from the network.

Here we are less concerned about the commit behaviours of a single a user, and some of the analysis methods we will use will ignore them, so we will opt for removing them here. However, this is something to consider exploring further as an optional "Extra:" task in independent study.

To remove the edges from the network we could either add an additional check during the building of the network, similar to the check about whether an edge already exists or we can remove all the edges afterwards. We will do the later for simplicity and use the following NetworkX DiGraph function which can be called from any graph object:

`G.remove_edges_from(..)`
https://networkx.org/documentation/stable/reference/classes/generated/networkx.DiGraph.remove_edg

The method requires a list of edges to remove from the network. Fortunately, NetworkX has a convenient method for getting the list of all edges that are self loops in a given Network:

`nx.selfloop_edges(..)`
https://networkx.org/documentation/stable/reference/generated/networkx.classes.function.selfloop_edg

So to remove all self loops from our network we can combine calls to these two calls as below:

In [7]:
```
G.remove_edges_from(nx.selfloop_edges(G))
```

> **Extra**: Modify the cell above to investigate and confirm that the operation does as intended. e.g., Print and compare the number of edges with the output of the earlier cell, re-visualise the network as a plot, etc.

Building multiple networks

The notebook so far has concentrated on building the network to model the commit behaviour of a single repository. However here we want to compare the behaviour (i.e., network structure) of multiple repositories. To do this we will define a new Python method that will take a csv file representing commits for a repository, create a network in the same way the previous cells above, and then return the created network from the method.

We will use a Python dictionary to store our networks together, the keys will be the names of the account owning the repository + the repository names (the same as the filenames for the csv files) and the values will be NetworkX DiGraph objects, e.g.

```
{
    "juputerlab+jupyterlab" : A DiGraph object,
    "microsoft+vscode" : A DiGraph object,
    ...
}
```

In [8]:
```python
# Define a Python function with one argument, the file path and name of the CSV file
def commitsToNetwork(fn):

    # Build the network as a NetworkX DiGraph for the given CSV file:

    # The following lines are the same as the code cell below the "Choosing a type of
    # Comments for each line are not present here, so refer back to the previous cell

    G = nx.DiGraph()

    with open(fn) as csvfile:
        reader = csv.reader(csvfile)

        user_of_the_next_commit = next(reader)[0]

        for i, commit in enumerate(reader):
            user_of_this_commit = commit[0]

            if G.has_edge(user_of_the_next_commit, user_of_this_commit):
                G[user_of_the_next_commit][user_of_this_commit]["weight"] += 1
            else:
                G.add_edge(user_of_the_next_commit, user_of_this_commit, weight = 1)

            user_of_the_next_commit = user_of_this_commit

    # Once the network has been built, remove any self loops that are present
    G.remove_edges_from(nx.selfloop_edges(G))

    # Return the completed network (DiGraph) object
    return G
```

In [9]:
```python
commitNetworks = {} # create the Python dictionary to hold our network objects

for filename in glob.glob("data/*.csv"): # for each csv file in the folder "data" (as

    repo = filename[5:-4] # this implementation is simple but inflexible to changes i
```

```
    # create the network DiGraph object for the csv file with that filename using the
    commitNetworks[repo] = commitsToNetwork(filename)
```

> **Extra**: The above cell uses the Python package 'glob' to do the heavy lifting in getting the file paths and names of the csv files we use. It is just one way of traversing file systems in Python. To learn more about the package, read the following article: https://towardsdatascience.com/the-python-glob-module-47d82f4cbd2d

Now that we have a dictionary of networks, one for each repository/csv file, lets loop over and print some basic network measures/statistics about each of the networks to give an early indication of the similarity/differences between them.

In [10]:
```
for repo, network in commitNetworks.items():  # for each item in the dictionary, prin
    print(f"Repository: {repo}\n"                  # the account+repository name, e.g
          f"#Nodes: {network.number_of_nodes()}   " # the number of nodes in the netw
          f"#Edges: {network.number_of_edges()}   " # the number of edges in the netw
          f"Total Weight: {network.size(weight='weight')}   " # the total weight on a
          f"Is Strongly Connected: {nx.is_strongly_connected(network)}\n" # whether t
    )
```

```
Repository: ipython+ipython
#Nodes: 89   #Edges: 180   Total Weight: 224.0   Is Strongly Connected: True

Repository: microsoft+vscode
#Nodes: 458   #Edges: 2339   Total Weight: 11255.0   Is Strongly Connected: True

Repository: v8+v8
#Nodes: 198   #Edges: 2465   Total Weight: 5442.0   Is Strongly Connected: True

Repository: openbsd+src
#Nodes: 106   #Edges: 1776   Total Weight: 5053.0   Is Strongly Connected: True

Repository: matplotlib+matplotlib
#Nodes: 205   #Edges: 877   Total Weight: 2733.0   Is Strongly Connected: True

Repository: numpy+numpy
#Nodes: 284   #Edges: 993   Total Weight: 2160.0   Is Strongly Connected: True

Repository: opencv+opencv
#Nodes: 267   #Edges: 711   Total Weight: 1256.0   Is Strongly Connected: True

Repository: scikit-learn+scikit-learn
#Nodes: 420   #Edges: 1054   Total Weight: 1259.0   Is Strongly Connected: True

Repository: golang+go
#Nodes: 375   #Edges: 1794   Total Weight: 3017.0   Is Strongly Connected: True

Repository: postgres+postgres
#Nodes: 26   #Edges: 302   Total Weight: 1538.0   Is Strongly Connected: True

Repository: nodejs+node
#Nodes: 377   #Edges: 1686   Total Weight: 2527.0   Is Strongly Connected: True

Repository: torvalds+linux
#Nodes: 4910   #Edges: 41237   Total Weight: 47446.0   Is Strongly Connected: True

Repository: rust-lang+rust
#Nodes: 992   #Edges: 7838   Total Weight: 15113.0   Is Strongly Connected: True

Repository: python+cpython
#Nodes: 526   #Edges: 1814   Total Weight: 2372.0   Is Strongly Connected: True

Repository: kubernetes+kubernetes
#Nodes: 890   #Edges: 3878   Total Weight: 6485.0   Is Strongly Connected: False

Repository: microsoft+TypeScript
#Nodes: 169   #Edges: 750   Total Weight: 1806.0   Is Strongly Connected: True

Repository: apple+swift
#Nodes: 271   #Edges: 3484   Total Weight: 13523.0   Is Strongly Connected: True

Repository: facebook+react
#Nodes: 161   #Edges: 425   Total Weight: 805.0   Is Strongly Connected: True

Repository: tidyverse+ggplot2
#Nodes: 60   #Edges: 117   Total Weight: 152.0   Is Strongly Connected: False

Repository: pytorch+pytorch
#Nodes: 968   #Edges: 6501   Total Weight: 8008.0   Is Strongly Connected: True

Repository: rstudio+rstudio
#Nodes: 40   #Edges: 363   Total Weight: 3486.0   Is Strongly Connected: True

Repository: nginx+nginx
#Nodes: 14   #Edges: 32   Total Weight: 58.0   Is Strongly Connected: False
```

```
Repository: scipy+scipy
#Nodes: 210   #Edges: 857   Total Weight: 1533.0   Is Strongly Connected: True

Repository: chromium+chromium
#Nodes: 2441   #Edges: 74552   Total Weight: 107281.0   Is Strongly Connected: True

Repository: tensorflow+tensorflow
#Nodes: 982   #Edges: 12695   Total Weight: 23528.0   Is Strongly Connected: False

Repository: freebsd+freebsd-src
#Nodes: 193   #Edges: 2906   Total Weight: 5383.0   Is Strongly Connected: True

Repository: apache+httpd
#Nodes: 21   #Edges: 110   Total Weight: 304.0   Is Strongly Connected: True

Repository: llvm+llvm-project
#Nodes: 1464   #Edges: 20999   Total Weight: 30928.0   Is Strongly Connected: True

Repository: gcc-mirror+gcc
#Nodes: 361   #Edges: 3639   Total Weight: 7098.0   Is Strongly Connected: True

Repository: denoland+deno
#Nodes: 382   #Edges: 1215   Total Weight: 2025.0   Is Strongly Connected: True

Repository: jupyterlab+jupyterlab
#Nodes: 126   #Edges: 620   Total Weight: 1673.0   Is Strongly Connected: True

Repository: apache+spark
#Nodes: 341   #Edges: 1807   Total Weight: 2666.0   Is Strongly Connected: True

Repository: angular+angular
#Nodes: 343   #Edges: 1377   Total Weight: 2273.0   Is Strongly Connected: True

Repository: pandas-dev+pandas
#Nodes: 607   #Edges: 1892   Total Weight: 3251.0   Is Strongly Connected: True
```

Some observations:

- The number of users committing the repositories vary considerably across the networks;
- As does the number of edges and edge weight;
- Most of the networks are strongly connected, but there are a few exceptions - e.g. the final node/user added to the network does not commit after another node/user.

Going forward this motivates caution in comparing the structural properties of the networks further and drawing conclusions on the similarities and differences found and how these may link to collaboration behaviour. To help overcome this we will compare the networks using various methods that are independent of the number of nodes and edges in the network (i.e., normalised).

---

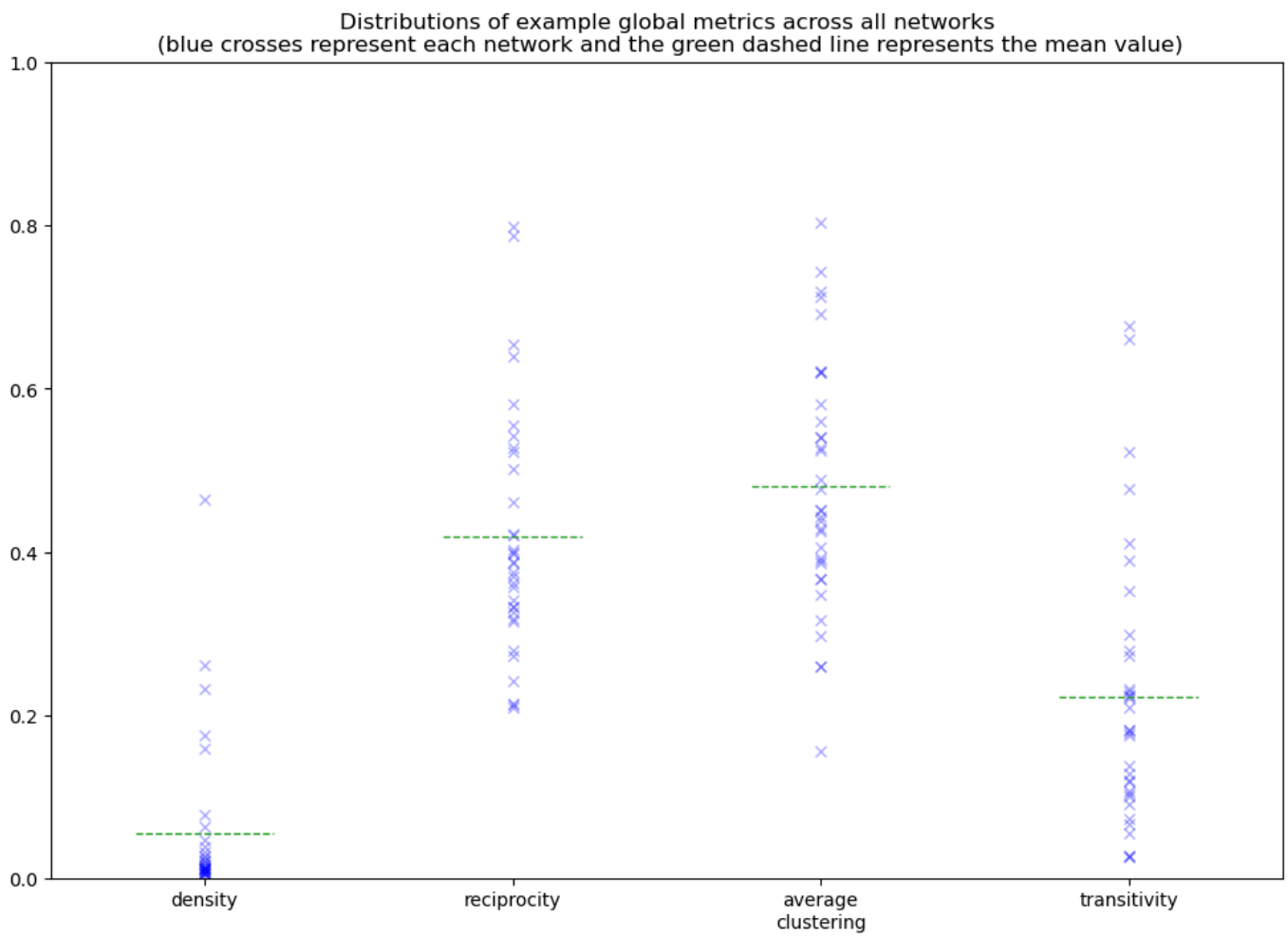
2. How similar/different is the commit behaviour across repositories?

Lets firstly utilise network measures that we've seen and used before in other notebooks that are 'global' in measuring/summarising something across the network and are independent of the number of nodes and edges.

Size-independent, global network measures.

In [11]:
```
# Create a Python list of lists where each sub-list contains the values of a particul

measures = [[nx.density(network) for network in commitNetworks.values()],
            [nx.reciprocity(network) for network in commitNetworks.values()],
            [nx.average_clustering(network) for network in commitNetworks.values()],
            [nx.transitivity(network) for network in commitNetworks.values()]]
```

In [12]:
```
fig1, ax1 = plt.subplots(figsize=(12,8))
ax1.set_title("Distributions of example global metrics across all networks\n (blue cr
ax1.set_ylim(0,1)

# Create a boxplot but turn off all visual features other than the mean line
bp = ax1.boxplot(
    x=measures,
    tick_labels=["density", "reciprocity", "average\nclustering", "transitivity"],
    showfliers=False,
    showbox=False,
    showcaps=False,
    whis=0,
    showmeans=True,
    meanline=True,
    medianprops=dict(linewidth=0)
)

# Plot the distribution of the data on top of the (invisible) boxplots
for i in range(0,4):
    y_values = measures[i]
    x_values = np.random.normal(i+1, 0.0, size=len(y_values))
    ax1.plot(x_values, y_values, 'bx', alpha=0.3)

plt.show()
```



> **Extra**: Transitivity recap, see Twitch notebook for more information https://transportgeography.org/contents/methods/graph-theory-measures-indices/transitivity-graph/

Some observations:

- The majority of the networks are very sparse;
- However, reciprocity, local average clustering, and global transitivity vary considerably across the networks.

This suggests that from the perspective of the handful of individual networks measures used, the networks are arguably more structurally dissimilar than similar and this is therefore indicative of dissimilarity in collaboration behaviour as expressed from the specific analysis lens of by commit ordering.

However this only considers some specific perspectives of how the network structure can be analysed and represented. To explore this further, we can look at alternative perspectives such as paths:

In [13]:
```
#In some (outdated) NetworkX versions, the average shortest path lengths for directed
#In newer versions of NetworkX, a Python Exception is raised if the network is not st

#Calculate the average shortest path length for networks that are strongly connected.
sp = [nx.average_shortest_path_length(network) for network in commitNetworks.values()

print("Average shortest path length across the networks:\n")
```

```
print(f"Number of networks={len(sp)}/{len(commitNetworks)}\n"
      f"Mean={np.mean(sp):.2f}\n"
      f"Median={np.median(sp):.2f}\n"
      f"SD={np.std(sp):.2f}\n"
      f"Min={np.amin(sp):.2f}\n"
      f"Max={np.amax(sp):.2f}\n"
     )
```

```
Average shortest path length across the networks:

Number of networks=30/34
Mean=2.86
Median=2.92
SD=0.54
Min=1.56
Max=4.10
```

While not a size-independent network measure, the average shortest path across the networks is similar. This provides an initial indication that the networks may be driven to some extent by some similar characteristics of the underlying collaboration behaviour.

Sub-structures in Networks: Triads

It has been argued in the Network Science field that complex networks representing real-world phenomena (and not exclusive to social behaviour) are built from 'building blocks', small subgraphs containing 3 or more nodes, rather than strict dyadic edges between nodes.

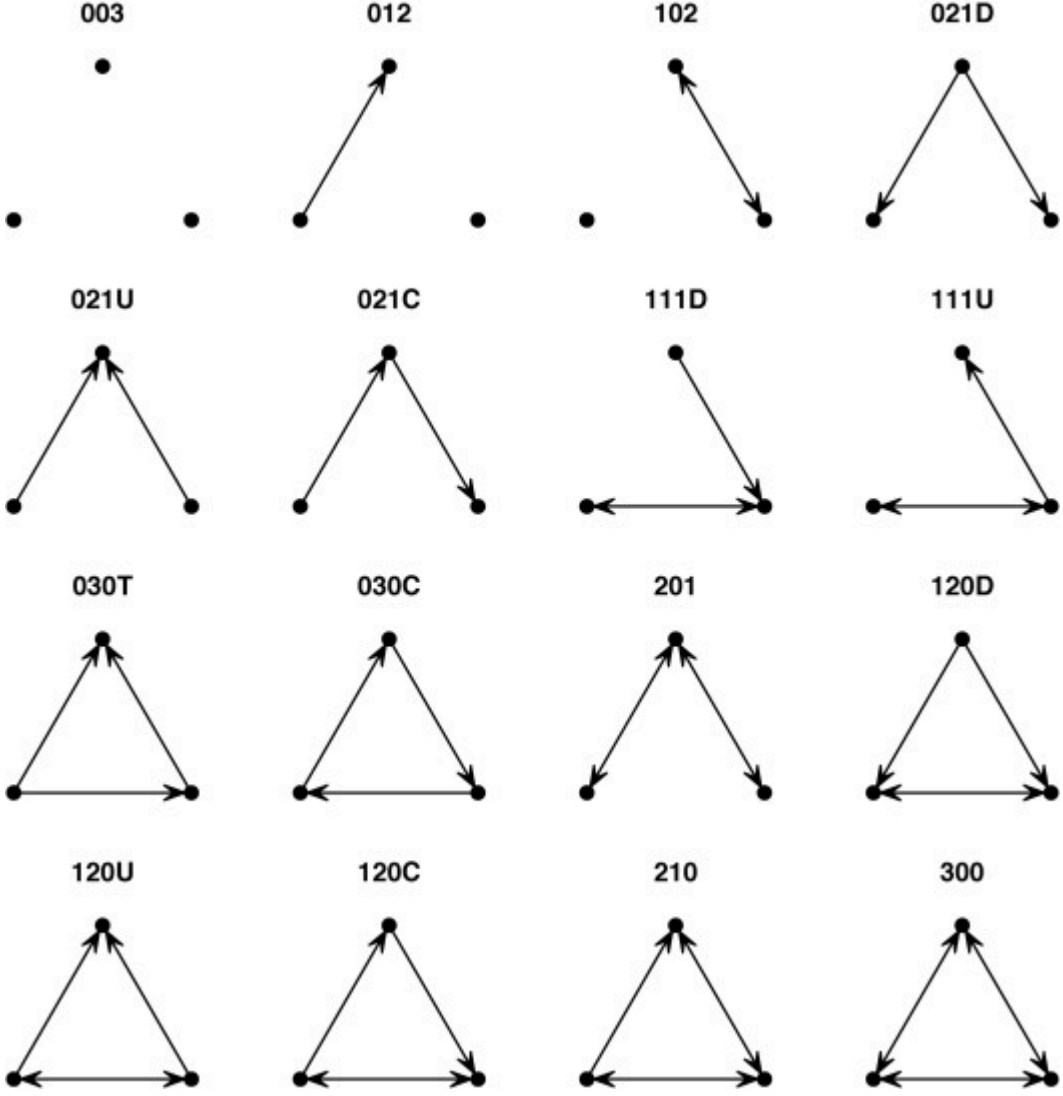
> **Extra**: Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., & Alon, U. (2002). Network motifs: simple building blocks of complex networks. Science, 298(5594), 824-827. http://sdcsb.ucsd.edu/wp-content/uploads/2013/10/Science-2002-Milo-824-7.pdf

and that 'superfamilies' of types of network structures can be seen when comparing the abundance or scarcity of particular building block subgraphs:

> **Extra**: Milo, R., Itzkovitz, S., Kashtan, N., Levitt, R., Shen-Orr, S., Ayzenshtat, I., ... & Alon, U. (2004). Superfamilies of evolved and designed networks. Science, 303(5663), 1538-1542. http://stat.asu.edu/~dieter/courses/APM_598/Alon_science.pdf

We've seen a small aspect of this in previous notebooks that highlighted the dominance of triangles (local clustering coefficient) in 'small-world' networks. To take this a step further, we will examine all of the possible ways that 3 nodes in a directed network can be connected together (triads) and then count how many of each occur in the GitHub Commit networks, i.e., create a 'census' of triads that exist in each network. If the proportion of each triad is similar then we will have some evidence that GitHub commit behaviour networks are comprised of specific building blocks.

Firstly, lets consider all of the possible ways that 3 nodes in a network can be connected (or not connected):



- Lienert et al

> The labels above each triad can be interpreted as follows:
>
> https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.
>
> {m}{a}{n}{type} (for example: 111D, 210, 102)
>
> Here:
>
> {m} = number of mutual ties/edges
>
> {a} = number of assymmetric ties/edges
>
> {n} = number of null ties/edges
>
> {type} = a letter (takes U, D, C, T) corresponding to up, down, cyclical and transitive. This is only used for topologies that can have more than one form (eg: 021D and 021U).

Triadic census

NetworkX has a method, that will count the number of times each type of triad occurs in a given network, i.e., the 'triadic census':

`nx.triadic_census(..)`
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.triads.t

To demonstrate this, lets calculate the triadic census for the jupyterlab network:

In [14]:
```
# The following will return a dictionary where keys are the triad labs (defined in th
# and the values are the number of times the triads occur in the network.

nx.triadic_census(commitNetworks["jupyterlab+jupyterlab"])
```

Out[14]:
```
{'003': 280203,
 '012': 22900,
 '102': 13929,
 '021D': 573,
 '021U': 547,
 '021C': 1105,
 '111D': 1975,
 '111U': 2006,
 '030T': 28,
 '030C': 26,
 '201': 1467,
 '120D': 35,
 '120U': 54,
 '120C': 150,
 '210': 269,
 '300': 233}
```

In the output you will see that the following triad types occur much more frequently than the others: 003, 012, and 102. These three triads are special in that, from one perspective, they arguably capture **what edges are not in the network**.

To contextual this, we can consider the triadic census as something that simply looks at every combination of 3 nodes that exist in the network. In this case there are 126 nodes in our network, and mathematically there are 325,500 combinations of 3 nodes that can be made from this irrespective of whether edges exist between these combinations of 3 or not. The sum of the triadic census values will equal the number of combinations of 3 (i.e., in this case 325,500).

So what 003 is actually counting is the number of combinations of 3 nodes in the networks where no edges exist between them. Importantly, these nodes are not necessarily disconnected (with a degree of 0) but just that no edges exist between the 3 nodes in the combination. As the network is sparse (as seen from the density measure earlier), this number is unsurprisingly high.

Similarly, 012 is counting the number of combinations where one edge exists between 2 of the 3 nodes, but not the other node in the combination. Likewise for 102.

> **Extra**: Modify the cell above to check and confirm that the sum of the values in the triadic census is equal to the number of combinations of 3 that can be made from a set of 126 nodes (325,500).

Connected triads

However, it is common practice when comparing networks using small induced subgraphs such as triads to focus on "Connected" combinations and discard the other node combinations.

Lets define a Python method that will perform the NetworkX triadic census operation, remove the counts for 003, 012, and 102, and then normalise the remaining 13 values so that they sum to 1. This will give us a comparable, size-independent measure between networks of different number of nodes, edges, and triad occurrences.

In [15]:
```
def calculate_normalised_connected_triadic_census(H):

    # Calculate the triadic census
    tc = nx.triadic_census(H)

    # Remove the non-connected triads
    del tc["003"]
    del tc["012"]
    del tc["102"]

    # Normalise the values to be between 0 and 1
    factor = 1.0 / sum(tc.values())
    for k in tc:
        tc[k] = round(tc[k] * factor, 3) #round to 3dp for readability here

    return tc

calculate_normalised_connected_triadic_census(commitNetworks["jupyterlab+jupyterlab"]
```

Out[15]:
```
{'021D': 0.068,
 '021U': 0.065,
 '021C': 0.13,
 '111D': 0.233,
 '111U': 0.237,
 '030T': 0.003,
 '030C': 0.003,
 '201': 0.173,
 '120D': 0.004,
 '120U': 0.006,
 '120C': 0.018,
 '210': 0.032,
 '300': 0.028}
```

If you multiplied the values by 100.0, they would represent the percentage of all triad occurrences are a particular type of triad. For example, ~17% of triad occurrences in the jupyterlab network is triad 201.

Comparing normalised censuses

Lets calculate the normalised triadic census for all networks, rather than just 'jupyterlab+jupyterlab' in the cell above, to then compare how similar the values of each triad type are across the networks.

(It will take a little while to run.)

In [16]:
```
# Calculate normalised triadic censuses for each commit network

# Create an empty dictionary, keys will be repositories and values will be another di
normalised_triadic_censuses = {}

# For each of our commit networks
for repository, network in commitNetworks.items():

    # Calculate the normalised triadic census for the network
    normalised_triadic_censuses[repository] = calculate_normalised_connected_triadic_

#print(normalised_triadic_censuses["jupyterlab+jupyterlab"])
```

In [17]:
```
# Print all normalised traidic censuses
print(normalised_triadic_censuses)
```

{'ipython+ipython': {'021D': 0.067, '021U': 0.075, '021C': 0.161, '111D': 0.238, '111
U': 0.225, '030T': 0.001, '030C': 0.004, '201': 0.217, '120D': 0.0, '120U': 0.0, '120
C': 0.008, '210': 0.005, '300': 0.001}, 'microsoft+vscode': {'021D': 0.089, '021U': 0.
087, '021C': 0.177, '111D': 0.233, '111U': 0.237, '030T': 0.001, '030C': 0.001, '201':
0.104, '120D': 0.004, '120U': 0.004, '120C': 0.012, '210': 0.024, '300': 0.027}, 'v8+v
8': {'021D': 0.074, '021U': 0.072, '021C': 0.151, '111D': 0.201, '111U': 0.196, '030
T': 0.014, '030C': 0.006, '201': 0.1, '120D': 0.015, '120U': 0.016, '120C': 0.035, '21
0': 0.07, '300': 0.05}, 'openbsd+src': {'021D': 0.054, '021U': 0.048, '021C': 0.105,
'111D': 0.18, '111U': 0.192, '030T': 0.013, '030C': 0.005, '201': 0.121, '120D': 0.01
8, '120U': 0.021, '120C': 0.043, '210': 0.102, '300': 0.098}, 'matplotlib+matplotlib':
{'021D': 0.085, '021U': 0.074, '021C': 0.164, '111D': 0.229, '111U': 0.242, '030T': 0.
002, '030C': 0.001, '201': 0.161, '120D': 0.002, '120U': 0.002, '120C': 0.01, '210':
0.017, '300': 0.011}, 'numpy+numpy': {'021D': 0.122, '021U': 0.103, '021C': 0.232, '11
1D': 0.202, '111U': 0.222, '030T': 0.002, '030C': 0.002, '201': 0.076, '120D': 0.002,
'120U': 0.003, '120C': 0.011, '210': 0.015, '300': 0.01}, 'opencv+opencv': {'021D': 0.
072, '021U': 0.062, '021C': 0.142, '111D': 0.214, '111U': 0.243, '030T': 0.002, '030
C': 0.001, '201': 0.251, '120D': 0.001, '120U': 0.001, '120C': 0.004, '210': 0.006, '3
00': 0.001}, 'scikit-learn+scikit-learn': {'021D': 0.158, '021U': 0.147, '021C': 0.32
9, '111D': 0.148, '111U': 0.153, '030T': 0.005, '030C': 0.003, '201': 0.029, '120D':
0.002, '120U': 0.003, '120C': 0.011, '210': 0.008, '300': 0.003}, 'golang+go': {'021
D': 0.115, '021U': 0.114, '021C': 0.24, '111D': 0.194, '111U': 0.196, '030T': 0.008,
'030C': 0.003, '201': 0.054, '120D': 0.007, '120U': 0.007, '120C': 0.019, '210': 0.02
6, '300': 0.018}, 'postgres+postgres': {'021D': 0.022, '021U': 0.02, '021C': 0.053, '1
11D': 0.104, '111U': 0.119, '030T': 0.011, '030C': 0.002, '201': 0.186, '120D': 0.014,
'120U': 0.016, '120C': 0.048, '210': 0.194, '300': 0.211}, 'nodejs+node': {'021D': 0.1
12, '021U': 0.121, '021C': 0.244, '111D': 0.194, '111U': 0.179, '030T': 0.012, '030C':
0.005, '201': 0.073, '120D': 0.006, '120U': 0.006, '120C': 0.017, '210': 0.023, '300':
0.009}, 'torvalds+linux': {'021D': 0.155, '021U': 0.156, '021C': 0.316, '111D': 0.147,
'111U': 0.144, '030T': 0.009, '030C': 0.003, '201': 0.049, '120D': 0.003, '120U': 0.00
3, '120C': 0.007, '210': 0.007, '300': 0.001}, 'rust-lang+rust': {'021D': 0.091, '021
U': 0.095, '021C': 0.188, '111D': 0.222, '111U': 0.214, '030T': 0.005, '030C': 0.002,
'201': 0.134, '120D': 0.005, '120U': 0.004, '120C': 0.011, '210': 0.02, '300': 0.009},
'python+cpython': {'021D': 0.12, '021U': 0.132, '021C': 0.264, '111D': 0.194, '111U':
0.176, '030T': 0.006, '030C': 0.003, '201': 0.073, '120D': 0.004, '120U': 0.003, '120
C': 0.01, '210': 0.012, '300': 0.004}, 'kubernetes+kubernetes': {'021D': 0.082, '021
U': 0.075, '021C': 0.161, '111D': 0.212, '111U': 0.23, '030T': 0.002, '030C': 0.001,
'201': 0.225, '120D': 0.001, '120U': 0.001, '120C': 0.003, '210': 0.006, '300': 0.00
1}, 'microsoft+TypeScript': {'021D': 0.136, '021U': 0.138, '021C': 0.176, '111D': 0.16
5, '111U': 0.19, '030T': 0.026, '030C': 0.002, '201': 0.062, '120D': 0.013, '120U': 0.
016, '120C': 0.021, '210': 0.033, '300': 0.022}, 'apple+swift': {'021D': 0.066, '021
U': 0.07, '021C': 0.134, '111D': 0.222, '111U': 0.2, '030T': 0.009, '030C': 0.003, '20
1': 0.145, '120D': 0.011, '120U': 0.012, '120C': 0.025, '210': 0.06, '300': 0.042}, 'f
acebook+react': {'021D': 0.115, '021U': 0.122, '021C': 0.255, '111D': 0.201, '111U':
0.2, '030T': 0.002, '030C': 0.002, '201': 0.068, '120D': 0.002, '120U': 0.003, '120C':
0.013, '210': 0.01, '300': 0.007}, 'tidyverse+ggplot2': {'021D': 0.148, '021U': 0.132,
'021C': 0.308, '111D': 0.156, '111U': 0.178, '030T': 0.002, '030C': 0.006, '201': 0.04
2, '120D': 0.002, '120U': 0.0, '120C': 0.019, '210': 0.007, '300': 0.002}, 'pytorch+py
torch': {'021D': 0.148, '021U': 0.146, '021C': 0.303, '111D': 0.142, '111U': 0.145, '0
30T': 0.02, '030C': 0.007, '201': 0.032, '120D': 0.008, '120U': 0.008, '120C': 0.018,
'210': 0.019, '300': 0.005}, 'rstudio+rstudio': {'021D': 0.023, '021U': 0.015, '021C':
0.041, '111D': 0.166, '111U': 0.176, '030T': 0.003, '030C': 0.0, '201': 0.177, '120D':
0.01, '120U': 0.022, '120C': 0.027, '210': 0.106, '300': 0.234}, 'nginx+nginx': {'021
D': 0.067, '021U': 0.122, '021C': 0.289, '111D': 0.089, '111U': 0.211, '030T': 0.033,
'030C': 0.011, '201': 0.056, '120D': 0.011, '120U': 0.0, '120C': 0.056, '210': 0.044,
'300': 0.011}, 'scipy+scipy': {'021D': 0.107, '021U': 0.097, '021C': 0.214, '111D': 0.
203, '111U': 0.218, '030T': 0.005, '030C': 0.004, '201': 0.074, '120D': 0.005, '120U':
0.006, '120C': 0.02, '210': 0.027, '300': 0.019}, 'chromium+chromium': {'021D': 0.148,
'021U': 0.148, '021C': 0.299, '111D': 0.117, '111U': 0.121, '030T': 0.025, '030C': 0.0
09, '201': 0.095, '120D': 0.006, '120U': 0.006, '120C': 0.012, '210': 0.012, '300': 0.
002}, 'tensorflow+tensorflow': {'021D': 0.1, '021U': 0.102, '021C': 0.204, '111D': 0.1
84, '111U': 0.178, '030T': 0.018, '030C': 0.006, '201': 0.14, '120D': 0.008, '120U':
0.008, '120C': 0.017, '210': 0.027, '300': 0.008}, 'freebsd+freebsd-src': {'021D': 0.0
75, '021U': 0.077, '021C': 0.157, '111D': 0.191, '111U': 0.187, '030T': 0.018, '030C':
0.007, '201': 0.105, '120D': 0.017, '120U': 0.017, '120C': 0.037, '210': 0.071, '300':
0.039}, 'apache+httpd': {'021D': 0.037, '021U': 0.029, '021C': 0.11, '111D': 0.206, '1
11U': 0.194, '030T': 0.01, '030C': 0.005, '201': 0.159, '120D': 0.007, '120U': 0.012,

'120C': 0.037, '210': 0.12, '300': 0.074}, 'llvm+llvm-project': {'021D': 0.121, '021
U': 0.122, '021C': 0.247, '111D': 0.179, '111U': 0.178, '030T': 0.014, '030C': 0.005,
'201': 0.071, '120D': 0.008, '120U': 0.008, '120C': 0.017, '210': 0.023, '300': 0.00
8}, 'gcc-mirror+gcc': {'021D': 0.092, '021U': 0.091, '021C': 0.188, '111D': 0.199, '11
1U': 0.2, '030T': 0.014, '030C': 0.005, '201': 0.092, '120D': 0.011, '120U': 0.011, '1
20C': 0.026, '210': 0.046, '300': 0.025}, 'denoland+deno': {'021D': 0.114, '021U': 0.1
18, '021C': 0.244, '111D': 0.196, '111U': 0.198, '030T': 0.004, '030C': 0.003, '201':
0.089, '120D': 0.003, '120U': 0.002, '120C': 0.01, '210': 0.013, '300': 0.005}, 'jupyt
erlab+jupyterlab': {'021D': 0.068, '021U': 0.065, '021C': 0.13, '111D': 0.233, '111U':
0.237, '030T': 0.003, '030C': 0.003, '201': 0.173, '120D': 0.004, '120U': 0.006, '120
C': 0.018, '210': 0.032, '300': 0.028}, 'apache+spark': {'021D': 0.133, '021U': 0.123,
'021C': 0.265, '111D': 0.169, '111U': 0.177, '030T': 0.012, '030C': 0.006, '201': 0.04
2, '120D': 0.008, '120U': 0.009, '120C': 0.021, '210': 0.023, '300': 0.012}, 'angular+
angular': {'021D': 0.113, '021U': 0.105, '021C': 0.221, '111D': 0.2, '111U': 0.208, '0
30T': 0.007, '030C': 0.003, '201': 0.065, '120D': 0.006, '120U': 0.006, '120C': 0.019,
'210': 0.03, '300': 0.017}, 'pandas-dev+pandas': {'021D': 0.09, '021U': 0.113, '021C':
0.203, '111D': 0.239, '111U': 0.202, '030T': 0.002, '030C': 0.001, '201': 0.133, '120
D': 0.001, '120U': 0.001, '120C': 0.005, '210': 0.007, '300': 0.003}}

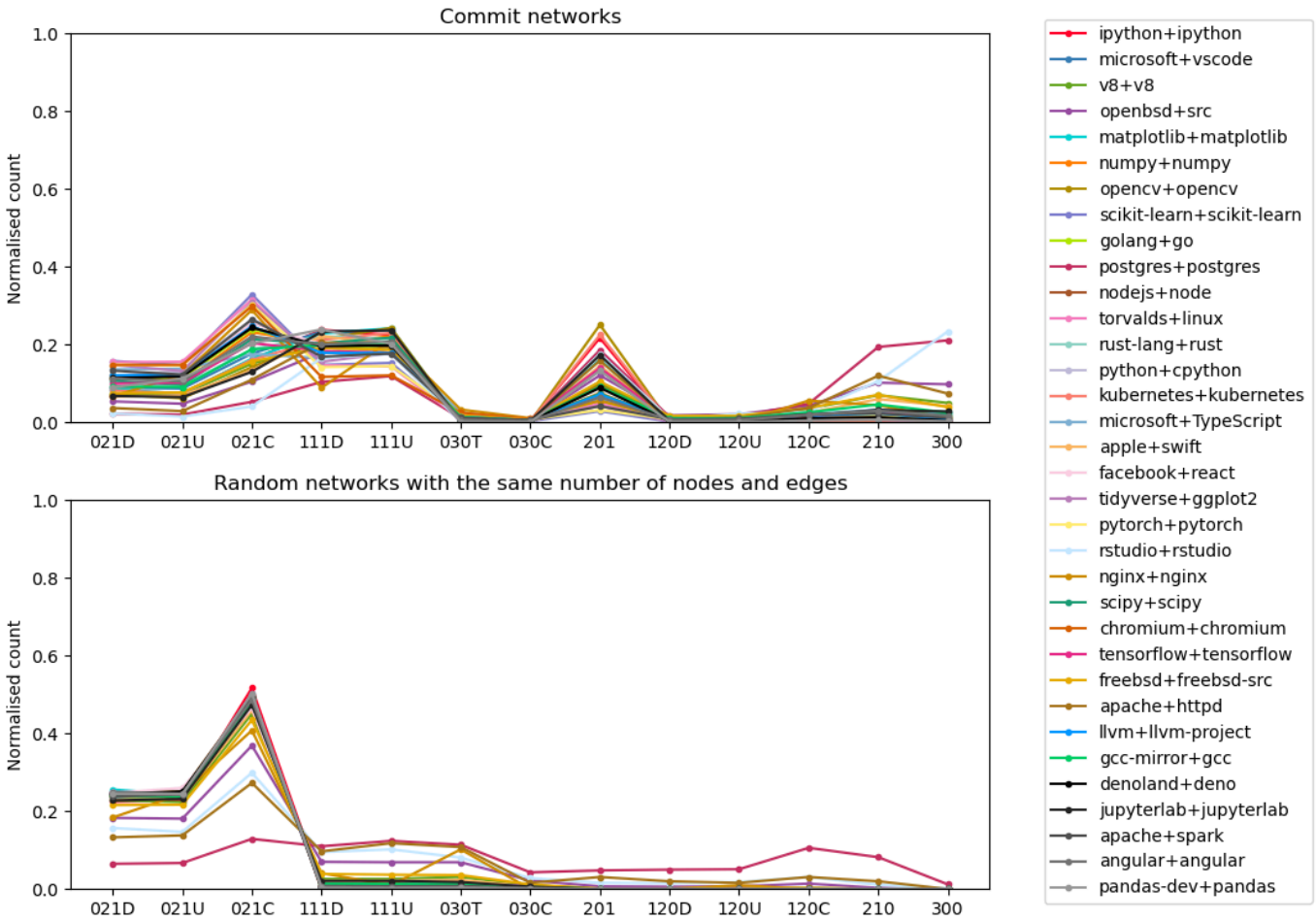
> **Extra**: As well as comparing the normalised triadic census between the commit networks, we can also compare what the normalised triadic census would be for an example random network with the same number of nodes and edges. Note, this doesn't ensure statistical robustness, but provides a demonstrative example.

In [18]:
```
# Calculate normalised triadic censuses for an example comparable, random network for

normalised_triadic_censuses_for_random_networks = {}

# For each of our commit networks, again
for repository, network in commitNetworks.items():

    # Generate a random, directed network with the same number of nodes and edges
    RN = nx.gnm_random_graph(
        network.number_of_nodes(),
        network.number_of_edges(),
        seed=1000,
        directed=True
    )

    # Calculate the normalised triadic census again but for the random network
    normalised_triadic_censuses_for_random_networks[repository] = calculate_normalise
```

Lets plot the data and visually compare the distributions across the networks:

In [19]:
```
# Create a vertically split plot canvas
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 9))

# Defines a set of colours to represent each repository/random network
cols = ["#ff0029", "#377eb8", "#66a61e", "#984ea3", "#00d2d5", "#ff7f00", "#af8d00",

# Loop over the normalised triadic censuses for each commit network and plot these on
i=0
for repository, triad_data in normalised_triadic_censuses.items():
    triads = triad_data.keys()
    normalised_counts = triad_data.values()
    ax1.plot(triads, normalised_counts, label=repository, marker=".", color=cols[i])
```

```
    i+=1

ax1.set_ylim([0, 1.0])
ax1.set_ylabel("Normalised count")
ax1.set_title("Commit networks")

# Compress the sub-plot to enable a legend to be added
box = ax1.get_position()
ax1.set_position([box.x0, box.y0, box.width * 0.8, box.height])

# Add a legend
ax1.legend(loc='center left', bbox_to_anchor=(1.05, -0.1))

# Repeat the above for the random networks, plotting the normalised triadic censuses
i=0
for repository, triad_data in normalised_triadic_censuses_for_random_networks.items():
    triads = triad_data.keys()
    normalised_counts = triad_data.values()
    ax2.plot(triads, normalised_counts, label=repository, marker=".", color=cols[i])
    i+=1

ax2.set_ylim([0, 1.0])
ax2.set_ylabel("Normalised count")
ax2.set_title("Random networks with the same number of nodes and edges")
box = ax2.get_position()
ax2.set_position([box.x0, box.y0, box.width * 0.8, box.height])


plt.show()
```



Summary of observations

- The profile of the normalised triadic census is arguably distinct with some types of triads being scarce across all networks (e.g., $030T$, $030C$, $120D$, $120U$) and others more abundant (e.g., $021C$, $111U$, 201);

- However there is some variance between the commit networks for some triad types (e.g., $021C$, 210, 300).
- The profiles between the commit networks and the random networks are arguably different overall, with some notable differences in triad types (e.g., $111D$, $111U$, $030T$, 300);
- However, there are some similarities in individual triad types being abundant or scarce (e.g., $021D$, $021U$, $021C$).

- In summary, the results provide some evidence that despite the notable differences seen in the normalised global network measures (e.g., reciprocity, transitivity, etc.), the differences are arguably reduced when considering the distribution of triadic 'building blocks' across the networks and the commit networks could be considered as being structurally similar from this perspective.

- However, this only provides a preliminary analysis and statistical robustness should be considered if this analysis is taken further (See Extra:).

> **Extra**: Statistical robustness
>
> To improve the evidence base for the analysis conducted, i.e., confidently determine the similarity of the commit networks in terms of triad abundance (network motifs) and scarcity (network anti-motifs), and whether commit networks are a 'superfamily' of complex networks or part of another, the following considerations should be made in any subsequent analysis:
>
> - Using an ensemble of (i.e., multiple) random networks, rather than just a single example;
> - Using additional 'Null models' for generating the random networks, rather than just the same number of nodes and edges;
> - Incorporating formulas for determining whether the abundance and sparsity of particular triads are significantly different to that of random networks, rather than only by human interpretation of a plot of the distribution.
>
> **Optional Extra**: Information on how to do the above can be found in the following (repeated from the Extra: cells earlier in this section of the Notebook):
>
> > Milo, R., Itzkovitz, S., Kashtan, N., Levitt, R., Shen-Orr, S., Ayzenshtat, I., ... & Alon, U. (2004). Superfamilies of evolved and designed networks. Science, 303(5663), 1538-1542.
>
> . http://stat.asu.edu/~dieter/courses/APM_598/Alon_science.pdf
>
> **Optional Extra**: Network 'Motifs' based on the abundance and scarcity of building blocks can also be seen for networks with a temporal dimension.
>
> > Temporal motifs: https://snap.stanford.edu/temporal-motifs/

---

3. Considering commit behaviour within repositories

Analyst interest in the decomposition and comparison of commit behaviour could be focused on individual repositories rather than across them. Next we will explore two example questions we *could ask* about the commit data. Starting with a particular, example repository and then expanding to analysis of each repository.

3.1 Is commit behaviour similar over time?

The commit data has timestamps which allows us to create networks for different subsets of commits that span different time periods. From this we can investigate whether the observed network structure is similar from month to month, or how the network structure grows and evolves over the course of the year. Here we will focus on the later, but the former is explored in the "Extra" section at the end of the notebook.

Network Building

We currently have directed networks that represent the cumulative commit data over 12 months. To provide a basis for comparing aspects of the growth of the network we can create another network that only contains behaviour from the beginning of the time period the data represents.

Lets define a new Python method that will be an extended version of the commitsToNetwork(..) method defined earlier in Section 1 of this notebook. We will then use this to create a network containing the commits for the month of January only.

In [20]:
```
# Code comment in the method below are limited to notable differences/additions from
# the previous commitsToNetwork(..) method.

def commitsToNetworkWithDateCutoff(fn, maxDate):   # this method has an extra paramet
    H = nx.DiGraph()

    with open(fn) as csvfile:
        reader = csv.reader(csvfile)

        user_of_the_next_commit = next(reader)[0]

        for i, commit in enumerate(reader):
            user_of_this_commit = commit[0]

            # NEW ADDITION
            timestamp = commit[1]
            dt_timestamp = datetime.strptime(timestamp, "%d/%m/%Y %H:%M:%S %z")
            if dt_timestamp >= maxDate:
                user_of_the_next_commit = user_of_this_commit
                continue

            if H.has_edge(user_of_the_next_commit, user_of_this_commit):
                H[user_of_the_next_commit][user_of_this_commit]["weight"] += 1
            else:
                H.add_edge(user_of_the_next_commit, user_of_this_commit, weight=1)

            user_of_the_next_commit = user_of_this_commit

    H.remove_edges_from(nx.selfloop_edges(H))
```

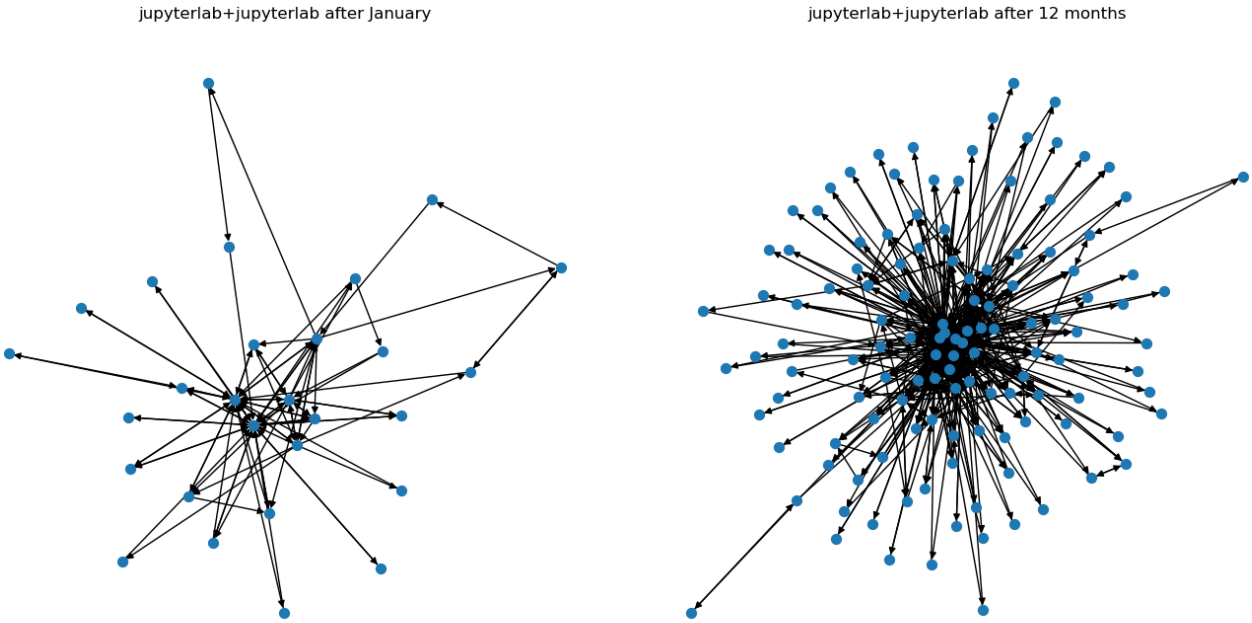
```
    return H


# Using the jupyterlab commit data as our example network
fn = "data/jupyterlab+jupyterlab.csv"

# Create a Python datetime object set as the 1st February
maxDate = datetime(2020, 2, 1, tzinfo=timezone.utc)

# Create a directed network of the JupyterLab data up to this maximum date (i.e., cre
J = commitsToNetworkWithDateCutoff(fn, maxDate)
```

In [21]:
```
print(f"Number of nodes: {J.number_of_nodes()}")
print(f"Number of edges: {J.number_of_edges()}")
print(f"Total of edge weights: {J.size(weight='weight')}")
```

```
Number of nodes: 28
Number of edges: 89
Total of edge weights: 177.0
```

The resulting January network has less edge weights as expected, but also less users than the network spanning 12 months (28 vs 126) and less edges between users (89 vs 677). Lets visualise the two networks before comparing the structures through network measures.

In [22]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))
nx.draw(J, ax=ax1, node_size=50)
nx.draw(G, ax=ax2, node_size=50)
ax1.set_title("jupyterlab+jupyterlab after January")
ax2.set_title("jupyterlab+jupyterlab after 12 months")
plt.tight_layout()
plt.show()
```



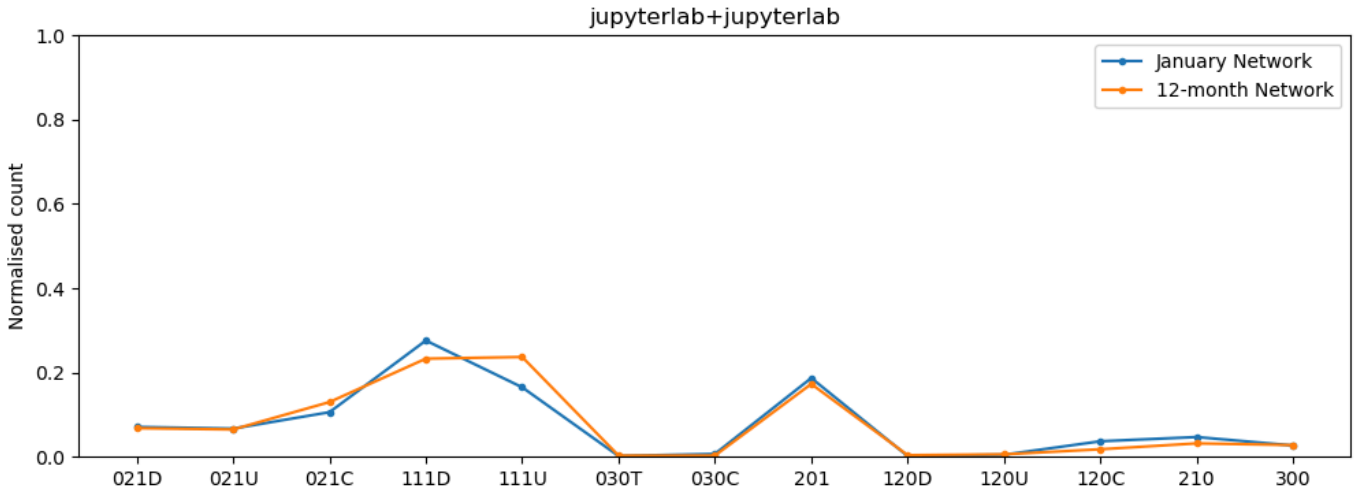
Triadic Census

Lets calculate the (normalised) triadic census for the January network and plot this against the 12-month network in a similar comparison to the task of comparing across networks of different repositories.

In [23]:
```
# Calculate the triadic census for J using the calculate_normalised_connected_triadic_
# method defined earlier and assign the result as J_census. Extract the 12-month netw
# existing normalised_triadic_censuses dictionary into a convenience variable, G_cens

J_census = calculate_normalised_connected_triadic_census(J)
```

```
G_census = normalised_triadic_censuses["jupyterlab+jupyterlab"]

# Plot the two censuses
```

In [24]:
```
# Plot the two censuses

fig, ax1 = plt.subplots(figsize=(12, 4))

# Plot the January and 12-month network on top of one another to allow a comparison
# on a per-triad basis as well as the overall distribution
ax1.plot(J_census.keys(), J_census.values(), label="January Network", marker=".")
ax1.plot(G_census.keys(), G_census.values(), label="12-month Network", marker=".")

ax1.set_ylim([0, 1.0])
ax1.set_ylabel("Normalised count")
ax1.set_title("jupyterlab+jupyterlab")

# Add a legend
ax1.legend(loc='upper right')

plt.show()
```



Some observations:

- The networks have very similar distributions;
- With some minor differences for $111D$ and $111U$;
- This suggests that the networks are built up by the same 'building blocks', despite representing different time periods, users, and commits.
- This also extends to the size-independent, global network measures also used earlier:

In [25]:
```
print(f"Density of J: {nx.density(J):.2f}\n"  # calculate the density of each network
      f"Density of G: {nx.density(G):.2f}\n"
)
print(f"Reciprocity of J: {nx.reciprocity(J):.2f}\n"  # calculate the overall recipro
      f"Reciprocity of G: {nx.reciprocity(G):.2f}\n"
)
print(f"Avg. Clustering of J: {nx.average_clustering(J):.2f}\n"  # calculate the aver
      f"Avg. Clustering of G: {nx.average_clustering(G):.2f}\n"
)
print(f"Transitivity of J: {nx.transitivity(J):.2f}\n"  # calculate the transitivity
      f"Transitivity of G: {nx.transitivity(G):.2f}\n"
)
```

```
Density of J: 0.12
Density of G: 0.04

Reciprocity of J: 0.58
Reciprocity of G: 0.58

Avg. Clustering of J: 0.53
Avg. Clustering of G: 0.58

Transitivity of J: 0.27
Transitivity of G: 0.23
```

Triadic Closure

As another temporally-driven example analyst task, lets examine whether the social phenomenon 'Triadic Closure' occurs over time for the example network.

> Triadic closure was first proposed by sociologist Georg Simmel, and can be defined as: given 3 nodes in a (social) network (e.g. with ids A, B, C), if a connection exists between A + B, and B + C, then over time there is a strong likelihood that a connection between A + C will form.

We've seen in previous notebooks that "small-world" social networks typically have high (local) clustering where a nodes friends are often friends themselves. This is an extension of that concept to consider this in a more dynamic network perspective.

Here we will use transitivity as an example network measure for triadic closure.

> Recap: NetworkX's transitivity method
> https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.

In [26]:
```
print(f"Transitivity of J: {nx.transitivity(J):.2f}\n"  # Calculate the transitivity
      f"Transitivity of G: {nx.transitivity(G):.2f}\n"  # Calculate the transitivity
)
```

```
Transitivity of J: 0.27
Transitivity of G: 0.23
```

The results does not support for triadic closure over time in the network.

However, this analysis is arguably biased as there are a different number of nodes in J and G. It could be that the users that arrive and commit after January have less transitivity than those that are present and commit in the first month.

To address this, lets repeat the analysis but only calculate the transitivity for the same users that appear in both J and G.

In [27]:
```
print(f"Transitivity of J: {nx.transitivity(J):.2f}\n"
      f"Transitivity in a subgraph of G with the same nodes as J: {nx.transitivity(G.
)
```

```
Transitivity of J: 0.27
Transitivity in a subgraph of G with the same nodes as J: 0.51
```

In this case there is evidence for triadic closure for those nodes in J between January and December.

To check whether this is common characteristic of the commit networks, or just for JupyterLab, lets repeat the triadic closure analysis for each repository:

In [28]:
```
for repository, network in commitNetworks.items():

    # Reconstruct the file location from the repository name, then build a January ne
    # using the commitsToNetworkWithDateCutoff(..) method defined earlier
    J = commitsToNetworkWithDateCutoff("data/"+repository+".csv", maxDate)

    # Extract an (induced) subgraph of the repository's 12-month network to only incl
    # also in the January network, J.
    G_with_only_J_nodes = network.subgraph(J.nodes())

    # Calculate the difference in transitivity from the January network to the 12-mon
    print(f"Repository: {repository} \n"
          f"Transitivity difference: {(nx.transitivity(G_with_only_J_nodes) - nx.tran
         )
```

```
Repository: ipython+ipython
Transitivity difference: 0.00

Repository: microsoft+vscode
Transitivity difference: 0.27

Repository: v8+v8
Transitivity difference: 0.33

Repository: openbsd+src
Transitivity difference: 0.28

Repository: matplotlib+matplotlib
Transitivity difference: 0.20
Repository: matplotlib+matplotlib
Transitivity difference: 0.20

Repository: numpy+numpy
Transitivity difference: 0.22

Repository: opencv+opencv
Transitivity difference: 0.09

Repository: scikit-learn+scikit-learn
Transitivity difference: 0.16

Repository: golang+go
Transitivity difference: 0.42

Repository: postgres+postgres
Transitivity difference: 0.46

Repository: nodejs+node
Transitivity difference: 0.29

Repository: torvalds+linux
Transitivity difference: 0.11

Repository: rust-lang+rust
Transitivity difference: 0.25

Repository: python+cpython
Transitivity difference: 0.22

Repository: kubernetes+kubernetes
Transitivity difference: 0.08

Repository: microsoft+TypeScript
Transitivity difference: 0.22

Repository: apple+swift
Transitivity difference: 0.28

Repository: facebook+react
Transitivity difference: 0.25

Repository: tidyverse+ggplot2
Transitivity difference: 0.15

Repository: pytorch+pytorch
Transitivity difference: 0.30

Repository: rstudio+rstudio
Transitivity difference: 0.31

Repository: nginx+nginx
```

```
Transitivity difference: 0.38

Repository: scipy+scipy
Transitivity difference: 0.37

Repository: chromium+chromium
Transitivity difference: 0.12

Repository: tensorflow+tensorflow
Transitivity difference: 0.25

Repository: freebsd+freebsd-src
Transitivity difference: 0.34

Repository: apache+httpd
Transitivity difference: 0.50

Repository: llvm+llvm-project
Transitivity difference: 0.19

Repository: gcc-mirror+gcc
Transitivity difference: 0.26

Repository: denoland+deno
Transitivity difference: 0.22

Repository: jupyterlab+jupyterlab
Transitivity difference: 0.24

Repository: apache+spark
Transitivity difference: 0.26

Repository: angular+angular
Transitivity difference: 0.39

Repository: pandas-dev+pandas
Transitivity difference: 0.16
```

Summary of observations

- We've seen how analysis of triads occurring in commit networks can also reveal similarities between networks built from commit data spanning different time periods.
- We've also seen that triadic closure is common between the GitHub users committing to repositories.
- However we should be careful about what we conclude from this. We have found some evidence in commit behaviours that *correlates* with what has been seen with triadic closure in social network structures. That does not necessarily mean that GitHub collaboration behaviours mimics exactly how social networks grow over time.

> **Extra**: Repeat the analysis in the cell above to use the average (local) clustering coefficient instead of transitivity and observe the extent that triadic closure occurs.

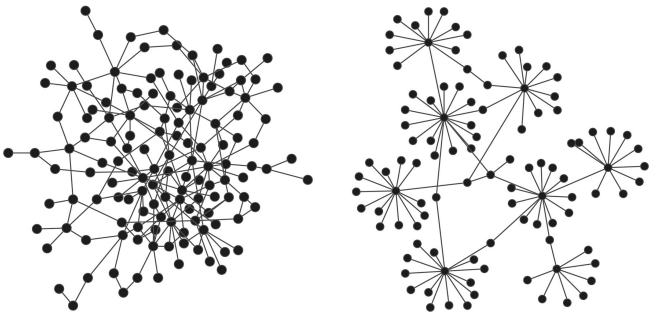
3.2 Are 'similar' users connected to each other?

So far we've largely focused on the structural characteristics of networks built from commit data from the perspective of sets of (3) nodes. We've seen how structural similarity between and within repositories can be seen through some, but not all network measures, despite differences in the number of users, edges, commits between the repositories.

We will move on with our investigations to examine whether the manner in which nodes (GitHub users) in the networks are connected together correlates with different node-to-node similarity phenomena seen in social and other complex networks more generally.

Firstly, a common characteristic of social networks is that there are 'hub' nodes with lots of connections. In undirected friendship social networks, this could be nodes that are very friendly and popular. In directed social networks, such as online communities like Twitter, Twitch, etc., this could be nodes with lots of 'in' edges (e.g., lots of followers).

In social networks, 'hubs' tend to have ties to other 'hubs', for example, lots of people know celebrities, but celebrities tend to know each other more than everyone that knows of them. This phenomenon doesn't occur in all types of real-world complex networks, for example in networks built to represent the protein interaction map of yeast, hubs typically avoid hubs.



The commit networks are arguably interesting to explore in this regard as they about collaborative connections between people but not necessarily in a communication 'social' sense that drives social media.

Secondly, another phenomena that occurs in social behaviour is the concept of "homophily", where 'similar' people tend to connect together and can even have similar lifestyles (see Extra, below). We see this in the typical human nature of forming groups based on shared interests, in both online and offline communities.

From a network perspective, both of these can be investigated through the concept of "assortativity" or "disassortativity". NetworkX provides several API methods depending on what type of data the attributes of nodes are representing:

https://networkx.org/documentation/stable/reference/algorithms/assortativity.html

> **Optional Extra**: Homophily in what venues people visit using Location Based Social Network data. Chorley, M. J., Whitaker, R. M., & Allen, S. M. (2015). Personality and location-based social networks. Computers in Human Behavior, 46, 45-56. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.667.7234

An introductory look at the connectivity between 'hubs'

Extra: Exploring the nodes with the largest degrees:

Lets begin by examining the largest in+out degrees in each 12-month commit network and whether these are concentrated on a single 'hub' user or multiple users.

In [29]:
```
# For each network, calculate the (in+out) degree sequence and then sort this from la
# Refer back to the Reddit notebook for more information on node degrees for directed

for name, network in commitNetworks.items():
    degree_sequence = sorted(network.degree(), reverse=True, key=operator.itemgetter(
    print(name)
    print(degree_sequence[0:3]) # print the 3 nodes with the largest degree [('the us
    print()
```

```
ipython+ipython
[('281349', 99), ('281273', 13), ('280940', 12)]

microsoft+vscode
[('420451', 201), ('420452', 189), ('420400', 186)]

v8+v8
[('404752', 164), ('404761', 152), ('404772', 146)]

openbsd+src
[('72136', 138), ('72038', 114), ('72225', 105)]

matplotlib+matplotlib
[('114594', 155), ('174584', 141), ('114651', 139)]

numpy+numpy
[('174904', 174), ('74793', 157), ('32645', 135)]

opencv+opencv
[('64502', 327), ('64297', 49), ('64467', 38)]

scikit-learn+scikit-learn
[('160818', 107), ('166005', 101), ('160868', 81)]

golang+go
[('122285', 181), ('133000', 116), ('121815', 113)]

postgres+postgres
[('172697', 48), ('172695', 42), ('172671', 40)]

nodejs+node
[('122329', 211), ('404748', 136), ('36410', 116)]

torvalds+linux
[('274017', 1450), ('274031', 866), ('273399', 733)]

rust-lang+rust
[('62428', 935), ('62398', 338), ('62395', 288)]

python+cpython
[('172952', 281), ('75049', 134), ('75060', 125)]

kubernetes+kubernetes
[('13687', 1039), ('13618', 162), ('13638', 101)]

microsoft+TypeScript
[('283611', 78), ('283626', 77), ('417545', 72)]

apple+swift
[('29455', 331), ('29427', 191), ('109924', 189)]

facebook+react
[('184564', 89), ('184541', 76), ('184517', 66)]

tidyverse+ggplot2
[('118411', 43), ('118407', 32), ('118414', 23)]

pytorch+pytorch
[('170514', 355), ('170491', 295), ('170497', 218)]

rstudio+rstudio
[('280615', 57), ('280695', 45), ('277747', 41)]

nginx+nginx
[('281484', 14), ('281475', 10), ('281482', 9)]
```

```
scipy+scipy
[('174913', 109), ('174910', 101), ('174746', 77)]

chromium+chromium
[('395572', 3147), ('395573', 1629), ('404772', 1130)]

tensorflow+tensorflow
[('159971', 1127), ('159966', 687), ('159276', 316)]

freebsd+freebsd-src
[('183472', 162), ('183482', 161), ('183453', 158)]

apache+httpd
[('33307', 28), ('33237', 25), ('33303', 19)]

llvm+llvm-project
[('109734', 818), ('109854', 623), ('109896', 552)]

gcc-mirror+gcc
[('188515', 223), ('133386', 219), ('133416', 187)]

denoland+deno
[('124655', 211), ('124654', 129), ('123598', 109)]

jupyterlab+jupyterlab
[('118188', 107), ('118118', 82), ('118077', 74)]

apache+spark
[('398401', 135), ('398429', 127), ('397645', 118)]

angular+angular
[('39755', 137), ('39762', 135), ('39523', 105)]

pandas-dev+pandas
[('31634', 513), ('4075', 144), ('4132', 105)]
```

From the above we can see that some networks have a single user with a notably large in+out degree and others have multiple users with similar, large in+out degrees (relative to the degree distribution in the individual network.

This leads to the question of whether these largely connected users are connected together themselves, or do they represent 'hubs' in different parts of the network?

One option to examine this is to explore and compare the sets of users these largely connected nodes are connected to. NetworkX provides methods for extracting this on a per-node basis:

```
G.neighbors(u)     # In NetworkX DiGraphs, this is the set of nodes that u
has an edge to  (i.e., the nodes that make up u's 'out' degree())
G.successors(u)    # In NetworkX DiGraphs, this is also the set of nodes
that u has an edge to  (i.e., the nodes that make up u's 'out' degree())
G.predecessors(u)  # In NetworkX DiGraphs, this is the set of nodes that
have an edge going to u (i.e., the nodes that make up u's 'in' degree())
```

https://networkx.org/documentation/stable/reference/classes/generated/networkx.DiGraph.neighbors.ht

https://networkx.org/documentation/stable/reference/classes/generated/networkx.DiGraph.successors.h

https://networkx.org/documentation/stable/reference/classes/generated/networkx.DiGraph.predecessor

As an example, lets examine the two users with the largest in+out degree for an example commit network.

In [30]:
```
# Assign the commit network to a convenience variable, H
H = commitNetworks["nginx+nginx"]

# Calculate the in+out degree sequence and then sort this from largest degree to smal
degree_sequence = sorted(H.degree(), reverse=True, key=operator.itemgetter(1))
print("Three nodes with the largest in+out degree:", dict(degree_sequence[:3]))

# Extract out the node ids for the nodes with the largest and second largest in+out d
node_with_largest_degree = degree_sequence[0][0]
node_with_second_largest_degree = degree_sequence[1][0]

# For the node with the largest in+out degree, extract out its 'neighbors', successor
# that are either a successor or a predecessor, as a second interpretation of 'neighb
ld_neighbors = set(H.neighbors(node_with_largest_degree))
ld_successors = set(H.successors(node_with_largest_degree))
ld_predecessors = set(H.predecessors(node_with_largest_degree))
ld_neighbors2 = ld_successors.union(ld_predecessors)

print(f"\n{node_with_largest_degree}'s 'neighbors':                {(', '.join(ld_nei
print(f"{node_with_largest_degree}'s neighbours on 'out' edges:  {(', '.join(ld_succe
print(f"{node_with_largest_degree}'s neighbours on 'in' edges:   {(', '.join(ld_prede
print(f"{node_with_largest_degree}'s neighbours on all edges:    {(', '.join(ld_neigh

# Repeat the above for the node with the second largest in+out degree
sld_neighbors = set(H.neighbors(node_with_second_largest_degree))
sld_successors = set(H.successors(node_with_second_largest_degree))
sld_predecessors = set(H.predecessors(node_with_second_largest_degree))
sld_neighbors2 = sld_successors.union(sld_predecessors)

print(f"\n{node_with_second_largest_degree}'s 'neighbors':                {(', '.join
print(f"{node_with_second_largest_degree}'s neighbours on 'out' edges:  {(', '.join(s
print(f"{node_with_second_largest_degree}'s neighbours on 'in' edges:   {(', '.join(s
print(f"{node_with_second_largest_degree}'s neighbours on all edges:    {(', '.join(s

# Print the nodes that are successors, predecessors, and neighbors for both nodes to

print(f"\nShared neighours (all edges):       {(', '.join(ld_neighbors2.intersection(
print(f"Shared neighbours on 'out' edges:   {(', '.join(ld_successors.intersection(sl
print(f"Shared neighbours on 'in' edges:    {(', '.join(ld_predecessors.intersection(
```

```
Three nodes with the largest in+out degree: {'281484': 14, '281475': 10, '281482': 9}

281484's 'neighbors':                281441, 281473, 281482, 281468, 281446, 281485, 2
81475, 281419
281484's neighbours on 'out' edges:  281441, 281473, 281482, 281468, 281446, 281485, 2
81475, 281419
281484's neighbours on 'in' edges:   281395, 281482, 281446, 281485, 281475, 281468
281484's neighbours on all edges:    281395, 281441, 281473, 281482, 281468, 281446, 2
81485, 281475, 281419

281475's 'neighbors':                281484, 281458, 281468, 281436
281475's neighbours on 'out' edges:  281484, 281458, 281468, 281436
281475's neighbours on 'in' edges:   281395, 281441, 281484, 281473, 281468, 281482
281475's neighbours on all edges:    281458, 281436, 281395, 281441, 281484, 281473, 2
81482, 281468

Shared neighours (all edges):       281395, 281441, 281473, 281468, 281482
Shared neighbours on 'out' edges:   281468
Shared neighbours on 'in' edges:    281468, 281395, 281482
```

Some observations:

- The two nodes with the largest in+out degrees have mutual edges between them;
- They also share some common neighbours, mostly through 'in edges', but not all;
- This provides some initial evidence that the 'hubs tend to have ties to other hubs' phenomenon may also apply to the commit networks.

Are users with similar degree connected?

To examine this at scale, the analysis can be generalised into measuring the 'assortativity' of the network - specifically the degree assortativity.

> Assortativity measures the similarity of connections in the graph with respect to the node degree - NetworkX documentation.

`nx.degree_assortativity_coefficient(..)`
The degree assortativity coefficient will be calculated as a value between -1 and +1. Where -1 suggests that the network is strongly disassortative (strong negative correlation between a node's degree and being connected, i.e., the hubs avoid each other, linking instead to small-degree nodes), +1 suggests that the network is strongly assortative (strong positive correlation between a node's degree and being connected together, i.e., hubs link to each other and avoid linking to small-degree nodes) and 0 suggests that the network has 'neutral' mixing.

https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.assorta

Lets calculate the degree assortativity coefficient for each (12-month) commit network and examine the descriptive statistics across them:

In [31]:
```
assortativity_per_commit_network = []

for repository, network in commitNetworks.items():                    # for each r
    deg_assortativity = nx.degree_assortativity_coefficient(network)     # calculate
    assortativity_per_commit_network.append(deg_assortativity)           # add it to
    print(f"Repository: {repository}\n"
          f"Degree Assortativity: {deg_assortativity:.2f}\n"           # and print
         )
```

Repository: ipython+ipython
Degree Assortativity: -0.41

Repository: microsoft+vscode
Degree Assortativity: -0.53

Repository: v8+v8
Degree Assortativity: -0.29

Repository: openbsd+src
Degree Assortativity: -0.30

Repository: matplotlib+matplotlib
Degree Assortativity: -0.55

Repository: numpy+numpy
Degree Assortativity: -0.41

Repository: opencv+opencv
Degree Assortativity: -0.36

Repository: scikit-learn+scikit-learn
Degree Assortativity: -0.22

Repository: golang+go
Degree Assortativity: -0.27

Repository: postgres+postgres
Degree Assortativity: -0.35

Repository: nodejs+node
Degree Assortativity: -0.25

Repository: torvalds+linux
Degree Assortativity: -0.10

Repository: rust-lang+rust
Degree Assortativity: -0.23

Repository: python+cpython
Degree Assortativity: -0.24

Repository: kubernetes+kubernetes
Degree Assortativity: -0.20

Repository: microsoft+TypeScript
Degree Assortativity: -0.30

Repository: apple+swift
Degree Assortativity: -0.38

Repository: facebook+react
Degree Assortativity: -0.48

Repository: tidyverse+ggplot2
Degree Assortativity: -0.50

Repository: pytorch+pytorch
Degree Assortativity: -0.14

Repository: rstudio+rstudio
Degree Assortativity: -0.40

Repository: nginx+nginx
Degree Assortativity: -0.41

```
Repository: scipy+scipy
Degree Assortativity: -0.37

Repository: chromium+chromium
Degree Assortativity: -0.11

Repository: tensorflow+tensorflow
Degree Assortativity: -0.19

Repository: freebsd+freebsd-src
Degree Assortativity: -0.25

Repository: apache+httpd
Degree Assortativity: -0.28

Repository: llvm+llvm-project
Degree Assortativity: -0.22

Repository: gcc-mirror+gcc
Degree Assortativity: -0.30

Repository: denoland+deno
Degree Assortativity: -0.31

Repository: jupyterlab+jupyterlab
Degree Assortativity: -0.41

Repository: apache+spark
Degree Assortativity: -0.26

Repository: angular+angular
Degree Assortativity: -0.27

Repository: pandas-dev+pandas
Degree Assortativity: -0.27
```

In [32]:
```
print("\nDescriptive Statistics:\n")
print(f"Mean degree assortativity: {np.mean(assortativity_per_commit_network):.2f}")
print(f"Std. Deviation: {np.std(assortativity_per_commit_network):.2f}\n")

print(f"Smallest degree assortativity: {np.amin(assortativity_per_commit_network):.2f
print(f"Largest degree assortativity: {np.amax(assortativity_per_commit_network):.2f}

print(f"Median degree assortativity: {np.median(assortativity_per_commit_network):.2f
```

```
Descriptive Statistics:

Mean degree assortativity: -0.31
Std. Deviation: 0.11

Smallest degree assortativity: -0.55
Largest degree assortativity: -0.10

Median degree assortativity: -0.30
```

Observations:

- The results indicate that, contrary to the expectation from looking at the 2 nodes with the largest in+out degree in the example network, the networks typically have moderate disassortative mixing or weak-disassortative-to-neutral mixing.

> **Extra**: The default NetworkX implementation of the degree_assortativity_coefficient(..) method compares the 'in' degree of source nodes with the 'out' degree of target nodes. Repeat the analysis above with different edge type combinations. Refer to the documentation page for the arguments to pass to the method to achieve this.

- However, as the network is a simple network where only one 'in edge' and 'out edge' can exist between nodes, the results may suffer from biasing effects such as "structural cutoffs". This consideration goes beyond the scope of this notebook, but more information can be found in optional, extended reading here:

  > **Optional Extra**: Structural Cutoffs
  > http://networksciencebook.com/chapter/7#structural-cutoffs

- Overall, **is this surprising given the dynamics of GitHub commits? Would we expect that users consciously wait to commit after other, particular users?**

Are users in the same timezone typically connected?

Another potentially correlating factor in the commit behaviour of nodes could be when during the day the commits happen. Lets investigate whether users committing from the same timezone are more likely to 'mix' than with other timezones. We will use an alternative NetworkX method for this, which will calculate the assortativity coefficient on a node attribute, rather than its degree:

`nx.attribute_assortativity_coefficient(..)`
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.assorta

Before we do this analysis we first have to determine and add a timezone attribute to the nodes. For simplicity at first, lets assume for now that users have a single, typical timezone and that this is the timezone of their last commit (i.e., first commit that appears in the time reverse sorted data).

Lets define a new Python method that will be an extended version of the commitsToNetwork(..) method defined earlier in Section 1 of this notebook.

In [33]:
```
# Code comment in the method below are limited to notable differences/additions from
# the previous commitsToNetwork(..) method.

def commitsToNetworkWithTimezone(fn):
    G = nx.DiGraph()

    # Create a Python dictionary to temporarily hold the map between node ids and a t
    timezones = {}
    with open(fn) as csvfile:
        reader = csv.reader(csvfile)

        user_of_the_next_commit = next(reader)[0]

        for i, commit in enumerate(reader):
            user_of_this_commit = commit[0]

            if G.has_edge(user_of_the_next_commit, user_of_this_commit):
                G[user_of_the_next_commit][user_of_this_commit]["weight"] += 1
```

```
            else:
                G.add_edge(user_of_the_next_commit, user_of_this_commit, weight=1)

            # Assume for simplicity for now that the last timezone a user is in is th
            # Using the timestamp column of the commit, split the string by blank spa
            # last list item (e.g., "31/12/2020 13:35:52 +0100" becomes ["31/12/2020"
            #                         becomes "+0100")
            if not user_of_this_commit in timezones:
                timezones[user_of_this_commit] = commit[1].split(" ")[-1]

            user_of_the_next_commit = user_of_this_commit

    G.remove_edges_from(nx.selfloop_edges(G))

    # Once the network is built and pruned of self-loops, add an attribute named "tz"
    # the value is the extracted timezone of their last commit, taken from the popula
    nx.set_node_attributes(G, timezones, "tz")
    return G



commitNetworksWithTimezone = {} # create the Python dictionary to hold our network ob

for filename in glob.glob("data/*.csv"): # for each csv file in the folder "data" (as

    repo = filename[5:-4] # this implementation is simple but inflexible to changes i

    commitNetworksWithTimezone[repo] = commitsToNetworkWithTimezone(filename)
```

In [34]:

```
tz_assortativity_per_commit_network = []

for repository, network in commitNetworksWithTimezone.items():
    tz_assortativity = nx.attribute_assortativity_coefficient(network, "tz")
    tz_assortativity_per_commit_network.append(tz_assortativity)

    print(f"Repository: {repository}\n"
          f"Timezone Assortativity: {tz_assortativity:.2f}\n"             # and print
         )
```

Repository: ipython+ipython
Timezone Assortativity: -0.13

Repository: microsoft+vscode
Timezone Assortativity: -0.01

Repository: v8+v8
Timezone Assortativity: 0.03

Repository: openbsd+src
Timezone Assortativity: nan

Repository: matplotlib+matplotlib
Timezone Assortativity: -0.02

Repository: numpy+numpy
Timezone Assortativity: -0.02

Repository: opencv+opencv
Timezone Assortativity: -0.06

Repository: scikit-learn+scikit-learn
Timezone Assortativity: 0.01

Repository: golang+go
Timezone Assortativity: -0.02

Repository: postgres+postgres
Timezone Assortativity: -0.05

Repository: nodejs+node
Timezone Assortativity: 0.00

Repository: torvalds+linux
Timezone Assortativity: 0.06

Repository: rust-lang+rust
Timezone Assortativity: -0.00

Repository: python+cpython
Timezone Assortativity: 0.01

Repository: kubernetes+kubernetes
Timezone Assortativity: 0.03

Repository: microsoft+TypeScript
Timezone Assortativity: -0.03

Repository: apple+swift
Timezone Assortativity: -0.03

Repository: facebook+react
Timezone Assortativity: -0.06

Repository: tidyverse+ggplot2
Timezone Assortativity: -0.01

Repository: pytorch+pytorch
Timezone Assortativity: 0.05

Repository: rstudio+rstudio
Timezone Assortativity: -0.03

Repository: nginx+nginx
Timezone Assortativity: 0.02

```
Repository: scipy+scipy
Timezone Assortativity: -0.01

Repository: chromium+chromium
Timezone Assortativity: -0.00

Repository: tensorflow+tensorflow
Timezone Assortativity: 0.01

Repository: freebsd+freebsd-src
Timezone Assortativity: -0.01

Repository: apache+httpd
Timezone Assortativity: nan

Repository: llvm+llvm-project
Timezone Assortativity: 0.06

Repository: gcc-mirror+gcc
Timezone Assortativity: 0.02

Repository: denoland+deno
Timezone Assortativity: -0.02

Repository: jupyterlab+jupyterlab
Timezone Assortativity: -0.01

Repository: apache+spark
Timezone Assortativity: -0.03

Repository: angular+angular
Timezone Assortativity: 0.01

Repository: pandas-dev+pandas
Timezone Assortativity: -0.02
```

```
/opt/homebrew/anaconda3/lib/python3.12/site-packages/networkx/algorithms/assortativit
y/correlation.py:282: RuntimeWarning: invalid value encountered in scalar divide
  r = (t - s) / (1 - s)
```

The Timezone attribute assortativity cannot be calculated for some repositories as they only contain users with the same timezone - resulting in nan values

In [35]:
```
print("\nDescriptive Statistics:\n")

print(f"Number of networks with calculatable timezone assortativity: {np.count_nonzer

print(f"Mean timezone assortativity: {np.nanmean(tz_assortativity_per_commit_network)
print(f"Std. Deviation: {np.nanstd(tz_assortativity_per_commit_network):.2f}\n")

print(f"Smallest timezone assortativity: {np.nanmin(tz_assortativity_per_commit_netwo
print(f"Largest timezone assortativity: {np.nanmax(tz_assortativity_per_commit_networ

print(f"Median timezone assortativity: {np.nanmedian(tz_assortativity_per_commit_netw
```

```
Descriptive Statistics:

Number of networks with calculatable timezone assortativity: 32

Mean timezone assortativity: -0.01
Std. Deviation: 0.04

Smallest timezone assortativity: -0.13
Largest timezone assortativity: 0.06

Median timezone assortativity: -0.01
```

Summary of observations

- The results suggest that the there is also little assortative or disassortative mixing on the basis of the timezone attribute used.

> **Optional Extra**: Consider whether a better label for the typical timezone of a user can be generated and whether this effects the observations and conclusions made.

---

4. Summary and motivations going forward

1. From the analysis, we've seen that despite wide variety in nodes, edges, and global network measures across networks built from the commit data, there are structural similarities between the networks when considering sub-structures (triads) and this extends to comparing networks within repositories that represent different time periods. This observation is similar to what has been found in social and other complex, 'real-world' networks more generally.

2. In comparison to phenomena often seen in social networks, the networks typically have triadic closure between users over time, but the connectivity between users show typical neutral mixing between users with different numbers of connections and timezones.

3. This suggests that the commit behaviour is likely driven by other factors than those that typically correlate in social networks. This is by no means a negative result, and is arguably unsurprising given that commits are not necessary a 'social' action between individuals but an action among individuals. It also highlights that networks of behaviour between people does not necessarily manifest into a structure that is akin to friend or follower social networks.

4. The results motivate further exploration of the networks, but this is beyond the scope of this notebook's main sections. The analysis extends with a demonstration of another means of comparing networks in the Extra section below. You may wish to extend the analysis of these GitHub commit networks further based on the content of other notebooks in the learning resources as well as further reading as part of independent study.

Along the way we have:

- Taken a further look at using Python and NetworkX to build different networks from data and then analyse them.
- We've looked at more methods from graph theory that, while applied to specific questions on the GitHub commit data, are generalisable to analysis of other directed networks (e.g., the Reddit Hyperlink networks in previous notebooks).
- Some we may use again in later weeks and the rest we could. Equally there may be more methods introduced in later weeks/notebooks where you can go back and implement in this notebook as an extension of this analysis. Therefore the analysis here should not be considered exhaustive for examining the data.

Extra: Monthly Commits and Graph Edit Distance

Another perspective to examine the commit data is to create networks representing each month and examine how 'stable' the commit behaviour patterns seen on a month-to-month basis.

Network Building

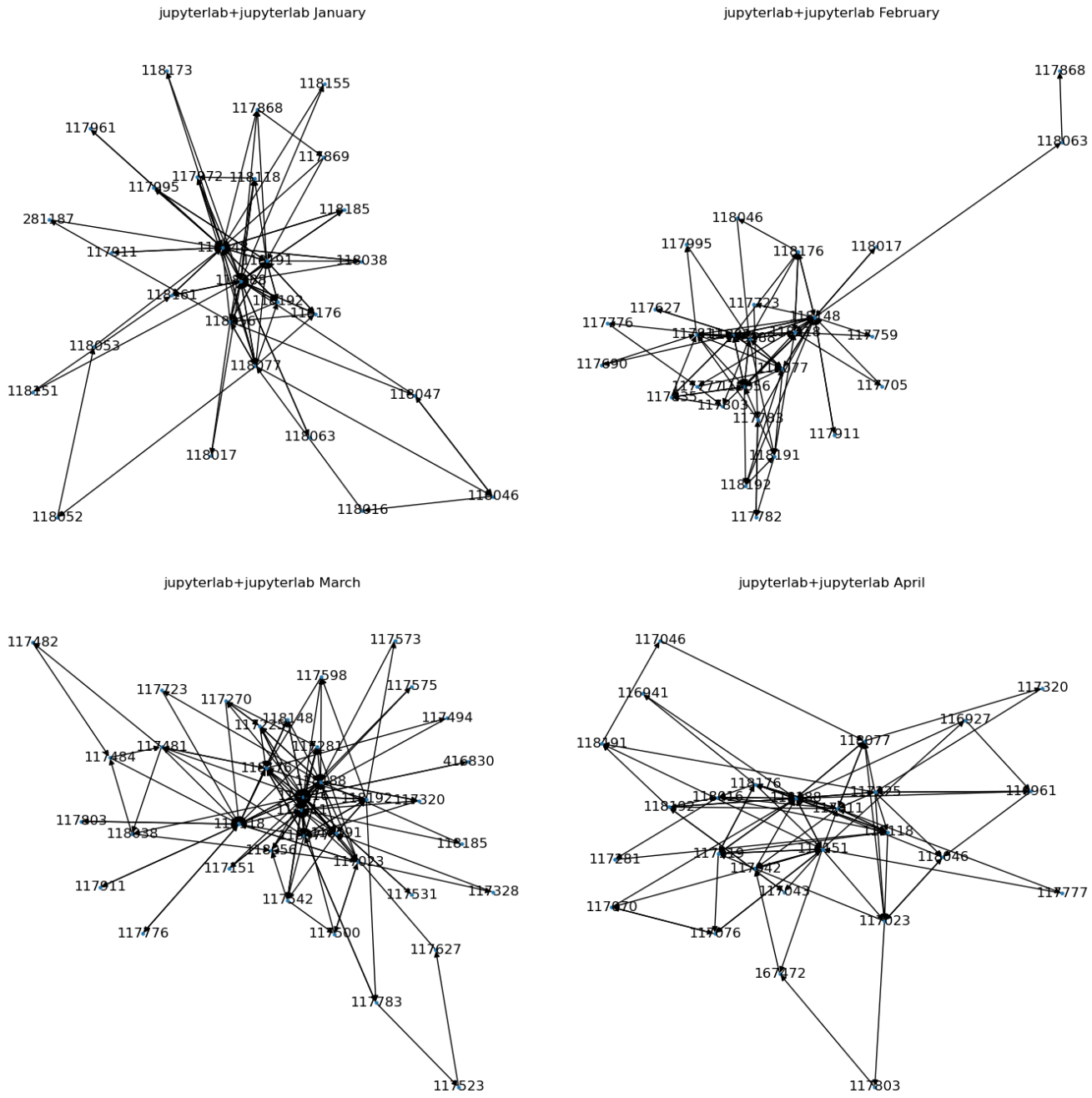
To examine this, lets create another alternative version of the commitsToNetwork(..) method defined in Section 1 of the notebook. Here we want, for a given repository of commit data, to create 12 networks, one for each month.

In [36]:
```
# There are multiple ways that this could be achieved.
# The below splits the method into two sections for simplicity,
# but it does mean looping over the commit data twice rather than once.

def commitsToNetworksByMonth(fn):

    # Create a Python dictionary to store the commit rows for each month
    commits_by_month = {str(month) : [] for month in range(1,13)}

    # For row in the csv file containing the commit data for a given repository
    with open(fn) as csvfile:
        reader = csv.reader(csvfile)

        # Loop over each commit and extract the timestamp column as a string
        # Convert the timestamp to a Python datetime object
        # Extract out the month as a string (e.g., 1, 2, 3, .. 12)
        # Using the month, add the commit row to the list of commits for the
        # given month in the dictionary created at the start of the method
        for i, commit in enumerate(reader):
            timestamp = commit[1]
            dt_timestamp = datetime.strptime(timestamp, "%d/%m/%Y %H:%M:%S %z")
            month = str(dt_timestamp.month)

            commits_by_month[month].append(commit)
```

```
    # Now that we have grouped the commit data by month, lets loop over the
    # commits for each month and create a directed network.

    # Create a dictionary to hold the directed networks
    G_for_each_month = {}

    # For each month, and list of commits
    for month, commits in commits_by_month.items():

        # Create a new DiGraph and assign it as the value for the month in the new di
        G_for_each_month[month] = nx.DiGraph()

        # Get the user in the first commit in the list
        user_of_the_next_commit = commits[0][0]

        # Starting from the next commit, loop over the commits and create the network
        # original commitToNetworks(..) method in Section 1.
        for i, commit in enumerate(commits[1:]):
            user_of_this_commit = commit[0]

            if G_for_each_month[month].has_edge(user_of_the_next_commit, user_of_this
                G_for_each_month[month][user_of_the_next_commit][user_of_this_commit]
            else:
                G_for_each_month[month].add_edge(user_of_the_next_commit, user_of_thi

            user_of_the_next_commit = user_of_this_commit

        G_for_each_month[month].remove_edges_from(nx.selfloop_edges(G_for_each_month[

    # Return the dictionary of 12 networks
    return G_for_each_month


# The above network can be used for any and all of the commit data files.
# However, lets continue focusing on JupyterLab as an example case study.
fn = "data/jupyterlab+jupyterlab.csv"
G_Months = commitsToNetworksByMonth(fn)
```

In [37]:

```
# Lets visualise a sample of the created networks

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(14, 14))

nx.draw(G_Months["1"], ax=ax1, node_size=5, with_labels=True)
nx.draw(G_Months["2"], ax=ax2, node_size=5, with_labels=True)
ax1.set_title("jupyterlab+jupyterlab January")
ax2.set_title("jupyterlab+jupyterlab February")

nx.draw(G_Months["3"], ax=ax3, node_size=5, with_labels=True)
nx.draw(G_Months["4"], ax=ax4, node_size=5, with_labels=True)
ax3.set_title("jupyterlab+jupyterlab March")
ax4.set_title("jupyterlab+jupyterlab April")

plt.tight_layout()
plt.show()
```



Edit distance

Another means of comparing how different two networks are to quantify what changes (i.e., in the nodes and edges) would need to be made to make one network the same as the other. This can include comparing the 'content' of the network in addition to its raw structure, i.e. where the attributes of nodes and edges are considered.

To do this we can use the concept of "Graph Edit Distance", where:

> Graph edit distance is a graph similarity measure analogous to Levenshtein distance for strings. It is defined as minimum cost of edit path (sequence of node and edge edit operations) transforming graph G1 to graph isomorphic to G2.- NetworkX Documentation

One method for assessing this would be to simply count the number of nodes that need to be removed, the number of missing nodes that need to be added, the number of edges to remove, and

the number of missing edges to add. However this may lead to some redundant operations, instead we could also consider a third type of operation 'substitute' to swap nodes and edges.

NetworkX provides several API methods for assessing the edit distance (https://networkx.org/documentation/stable/reference/algorithms/similarity.html), and most notably:

`nx.graph_edit_distance(..)`
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.similari

However, finding the exact edit distance between two graphs/networks is an NP-hard problem. NetworkX as a result is often quite slow to calculate the exact edit distance using the method above. Therefore, it also provides the ability to generate less-optimal 'solutions' which we can expect to be close to the exact edit distance, but not guaranteed.

Two notable methods for this are:

```
nx.optimize_graph_edit_distance(..)  # Generates edit distance 'solutions'
and returns as the total number add, remove, and substitute operations
```
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.similari

and

```
nx.optimize_edit_paths(..) # Generates edit distance 'solutions' and returns
the exact add, remove, and substitute operations in each
```
https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.similari

In an analyst task we may only be interested in the edit distance value. However, for demonstration purposes lets use the second method and examine a generated 'solution' for the operations that would be needed to convert, the commit network for January into the *isomorphic equivalent* of the commit network for February.

In this first example with the default parameters/arguments, it will ignore node and edge attributes (including node ids) and focus on the changes needed for the January network to become the same Graph structure as February network.

Note, in the default parameters, node and edge substitutions have a 'cost' of 0, i.e. they do not contribute to the edit distance. Whereas adding/removing nodes and edges do.

In [38]:
```
solutions = nx.optimize_edit_paths(G_Months["1"], G_Months["2"])

# the method returns a Python Generator object for efficiency, not a list

solution = next(solutions)   # move the generator to the first attempted solution

node_changes = solution[0]   # get the list of change operations needed to J's nodes
edge_changes = solution[1]   # get the list of change operations needed to J's edges
edit_distance = solution[2]  # get the edit distance of the solution (# of node chang

print("Edit distance (Cost): %d" % edit_distance)  # total number of node and edge op
print("Total number of node and edge operations: %d" % (len(node_changes)+len(edge_ch

print("\nExample node operations (from, to):")
for change in node_changes[:30]: #limit to the first 30 for visualisation purposes
    print(change)
    # 'None' values in the first position represents a node being added.
    # 'None' values in the second position represents a node being removed.
    # values in both positions represent a node substitution
```

```python
print("\nExample edge operations (from, to):")
for change in edge_changes[:30]: #limit to the first 30 for visualisation purposes
    print(change)
    # 'None' values in the first position represents an edge being added.
    # 'None' values in the second position represents an edge being removed.
    # values in both positions represent an edge substitution
```

```
Edit distance (Cost): 117
Total number of node and edge operations: 171

Example node operations (from, to):
('117869', None)
('118191', '118188')
('118056', '117627')
('117868', '118077')
('118188', '118148')
('118148', '117811')
('117995', '118056')
('118038', '117835')
('118118', '118176')
('117972', '118046')
('117911', '118118')
('118077', '118191')
('118192', '117690')
('117961', '117705')
('118176', '117911')
('118046', '117723')
('118016', '117783')
('118017', '118017')
('118047', '117759')
('118052', '117777')
('118053', '117803')
('281187', '117776')
('118063', '117782')
('118161', '118192')
('118151', '117995')
('118155', '118063')
('118185', '117868')
('118173', '118016')

Example edge operations (from, to):
(('117869', '118191'), None)
(('118191', '118056'), ('118188', '117627'))
(('118056', '118191'), ('117627', '118188'))
(('118191', '117868'), ('118188', '118077'))
(('117868', '117869'), None)
(None, ('118077', '118188'))
(('118191', '118188'), ('118188', '118148'))
(('118056', '118188'), None)
(('117868', '118188'), ('118077', '118148'))
(('118188', '118056'), None)
(('118188', '118191'), ('118148', '118188'))
(None, ('118148', '118077'))
(('117869', '118148'), None)
(('118191', '118148'), ('118188', '117811'))
(('118056', '118148'), None)
(('118188', '118148'), ('118148', '117811'))
(('118148', '117868'), ('117811', '118077'))
(('118148', '118188'), None)
(('118148', '118191'), None)
(('118191', '117995'), ('118188', '118056'))
(('118148', '117995'), ('117811', '118056'))
(('117995', '118191'), ('118056', '118188'))
(('117995', '118148'), None)
(None, ('118077', '118056'))
(None, ('118056', '118148'))
(('118191', '118038'), None)
(('118148', '118038'), None)
(('118038', '118148'), ('117835', '117811'))
(('118038', '118188'), None)
(None, ('118056', '117835'))
```

Some observations:

- The total number of operations (node and edge operations) is larger than the reported edit 'distance'. This is because NetworkX is calculating the edit distance as a 'cost' where node and edge additions/deletions have a cost but substitutions (operations without a 'None' in the first or second position) have a cost of 0.

We can loop over the Generator object to have NetworkX keep looking for 'better' solutions, where the edit distance/cost is reduced.

In [39]:
```
i=0
for solution in solutions:
    print(f"Discovered edit distance: {solution[2]}")
    i+=1
    if i==3: break
```

```
Discovered edit distance: 115.0
Discovered edit distance: 113.0
Discovered edit distance: 111.0
```

The default parameters/arguments of NetworkX's edit distance methods focus on the changes needed to make one network the isomorphic equivalent of another. Where node and edge attributes (including ids) are ignored and this is just the underlying graph structure of the networks.

However, we can change this so that, for example, the node ids between the networks are considered. In the case of the GitHub networks, we've seen that the networks for each month often have many shared nodes (where users continue to commit to the repository over the year the data spans).

To do this we need to add the node id as an *attribute* to the node as strictly speaking the node id is kept separate from a node's attributes (like the timezone attribute seen in Section 3 of this notebook). From this we then need to define a new method that NetworkX will use to compare if two nodes are similar or not, and then factor this into the list of operations needed to change one Network into the other.

This example represents a slight step forward in how the nx.optimize_graph_edit_distance(..) and nx.optimize_edit_paths(..) methods can be configured. You may wish to look at the documentation pages for how this can be extended further (e.g., assigning custom edit 'costs' to specific types of operations, defining a custom method for how edges are compared, etc.).

In [40]:
```
# Create a copy of the January network for convenience and avoid changing the origina
G1 = G_Months["1"].copy()
# Loop over each of the nodes and add the id of the node as an attribute with the key
for n in G1.nodes():
    G1.nodes[n]["label"] = n

# Repeat the above for the February network
G2 = G_Months["2"].copy()
for n in G2.nodes():
    G2.nodes[n]["label"] = n

# Define a custom method for NetworkX to compare whether two nodes are similar or not
def new_node_subst_cost(node1, node2):
    # check if the nodes have exactly the same value for the "label" attribute (which
    if node1["label"] == node2["label"]:
```

```
        return 0 # if the nodes have the same label/id, return an edit cost of 0
    return 1     # otherwise the edit cost is 1


solutions = nx.optimize_edit_paths(
    G1,
    G2,
    node_subst_cost=new_node_subst_cost  # set the node_subst_cost parameter to be th
)

solution = next(solutions)   # move the generator to the first attempted solution

node_changes = solution[0]   # get the list of change operations needed to J's nodes
edge_changes = solution[1]   # get the list of change operations needed to J's edges
edit_distance = solution[2]  # get the edit distance of the solution (# of node chang

print(f"Edit distance (Cost): {edit_distance}")  # total number of node and edge oper
print(f"Total number of node and edge operations: {(len(node_changes)+len(edge_change

print("Example node operations (from, to):")
for change in node_changes[:30]: #limit to the first 30 for visualisation purposes
    print(change)
    # 'None' values in the first position represents a node being added.
    # 'None' values in the second position represents a node being removed.
    # values in both positions represent a node substitution

print("\nExample edge operations (from, to):")
for change in edge_changes[:30]: #limit to the first 30 for visualisation purposes
    print(change)
    # 'None' values in the first position represents an edge being added.
    # 'None' values in the second position represents an edge being removed.
    # values in both positions represent an edge substitution
```

```
Edit distance (Cost): 125.0
Total number of node and edge operations: 169

Example node operations (from, to):
('117869', '117690')
('118191', '118191')
('118056', '118056')
('117868', '117868')
('118188', '118188')
('118148', '118148')
('117995', '117995')
('118038', '117627')
('118118', '118118')
('117972', '117811')
('117911', '117911')
('118077', '118077')
('118192', '118192')
('117961', '117835')
('118176', '118176')
('118046', '118046')
('118016', '118016')
('118017', '118017')
('118047', '117705')
('118052', '117723')
('118053', '117783')
('281187', '117759')
('118063', '118063')
('118161', '117777')
('118151', '117803')
('118155', '117776')
('118185', '117782')
('118173', None)

Example edge operations (from, to):
(('117869', '118191'), None)
(('118191', '118056'), ('118191', '118056'))
(('118056', '118191'), None)
(('118191', '117868'), None)
(('117868', '117869'), None)
(('118191', '118188'), None)
(('118056', '118188'), ('118056', '118188'))
(('117868', '118188'), None)
(('118188', '118056'), ('118188', '118056'))
(('118188', '118191'), ('118188', '118191'))
(('117869', '118148'), None)
(('118191', '118148'), None)
(('118056', '118148'), ('118056', '118148'))
(('118188', '118148'), ('118188', '118148'))
(('118148', '117868'), None)
(('118148', '118188'), ('118148', '118188'))
(('118148', '118191'), None)
(('118191', '117995'), None)
(('118148', '117995'), None)
(('117995', '118191'), None)
(('117995', '118148'), None)
(None, ('117995', '118188'))
(('118191', '118038'), None)
(('118148', '118038'), None)
(('118038', '118148'), None)
(('118038', '118188'), ('117627', '118188'))
(None, ('118188', '117627'))
(('118191', '118118'), ('118191', '118118'))
(('118056', '118118'), ('118056', '118118'))
(('118188', '118118'), ('118188', '118118'))
```

Again, we can loop over the Generator object to have NetworkX keep looking for 'better' solutions, where the edit distance/cost is reduced.

In [41]:
```
i=0
for solution in solutions:
    print(f"Discovered edit distance: {solution[2]}")
    i+=1
    if i==3: break
```

```
Discovered edit distance: 121.0
Discovered edit distance: 120.0
Discovered edit distance: 119.0
```

Some observations:

- The edit distance when considering the node ids in the similarity actually increases the edit distance in comparison to just the isomorphic equivalent edit cost.
- Many of the node 'substitutions' have the same node id for both values, showing the effect of the new_node_subst_cost(..) method.

> **Extra**: Create networks for each month for each repository (rather than just January) and explore how the structure and 'content' of the commit behaviour networks for each repository change over time. For example:
>
> 1. Compare the 12 networks for each repository using a triadic census in the same way as the comparisons across repositories in Part 2 of this notebook.
> 2. Compare the 12 networks for each repository using edit distance in the same way as between January and all months in Part 3 of this notebook.