How to Create a Time Series Network Graph Visualization in Python

Use Plotly and NetworkX to show how a network evolves over time

Claudia Ng

.

Published in

Towards Data Science

.

9 min read

.

Oct 27

In this article, you will learn to create a time series network visualization in Python that shows how connections in a network develop over time, as illustrated in the animation above. Network data is very effective for revealing connections, and time series data can be useful for uncovering trends and anomalies in the underlying data.

For this article, we will create an example using a Kaggle dataset on healthcare provider fraud. (This dataset is currently licensed CC0: Public Domain on Kaggle. Please note that this dataset might not be accurate, and it's used in this article only for demonstration purposes).

We will combine data from the provided link to get a cluster of fraudulent claims associated with the given attending physician, followed by plotting the connections between this physician and other entities (up to a certain number of hops away) over time based on the claim start date.

Be sure to have Plotly and NetworkX installed in your Python virtual environment.

If you want to learn an effective way to visualize how a network grows over time, read on!

Findings from Dataset

This dataset contains a total of 82,063 distinct physicians, of which 20,592 have at least one fraudulent claim. While the majority of these physicians have only a handful of fraudulent claims, there is a small fraction who are prolific fraudsters.

The top 25% of physicians have at least 5 claims, and the most egregious example is a physician with 2,534 fraudulent claims!

While it is easy to calculate these statistics on the physician level, a time series network visualization is a good way to make sense of the scale of fraud committed by this physician and over what timeframe.

Finding Connections

Given a physician, what if we could find its connections that also have fraudulent claims up to a certain number of hops away?

For this, we will write a function that takes a DataFrame with claims data, AttendingPhysician (string) and MaxHops (integer) as arguments and returns a DataFrame containing fraudulent claims associated with the given AttendingPhysician up to a maximum number of hops away.

Starting from one physician, this function executes to identify beneficiaries and physicians with fraudulent claims connected to the cluster. It continues as long as the number of iterations is at or under MaxHops and it hasn't converged, meaning that the cluster size at the end of the previous iteration is smaller than the cluster size after this iteration.

Below is a code snippet illustrating how to do this:

```
def get_fraud_cluster(df, AttendingPhysician, MaxHops):
    """
    Returns fraudulent claims associated with the given AttendingPhysician
    and beneficiaries served up to a maximum number of hops or convergence

        Parameters:
            df (DataFrame): A Pandas DataFrame with claims data.
            AttendingPhysician (str): ID of attending physician to start with as
the root node.
            MaxHops (int): Maximum number of hops.

        Returns:
```

```
            df (DataFrame): A Pandas Dataframe containing fraudulent claims
associated
            with supplied AttendingPhysician up to a maximum number of hops
away.
    """
    # Initial variables
    prev_cluster_size = 0
    current_cluster_size = 1
    i = 0

    # Add initial physician to set_physicians
    set_physicians = set([AttendingPhysician])
    set_beneficiaries = set()

    while prev_cluster_size < current_cluster_size and i < MaxHops:

        prev_cluster_size = len(set_physicians) + len(set_beneficiaries)

        # Get fraudulent physicians with claims associated with fraudulent
beneficiaries
        fraud_physicians = df[
            (df.BeneID.isin(set_beneficiaries)) &
            (df.PotentialFraud == 1)
        ]["AttendingPhysician"].unique()
        new_fraud_physicians = set(fraud_physicians).difference(set_physicians)

        # Update set with new fraudulent physicians
        set_physicians.update(new_fraud_physicians)

        # Get fraudulent beneficiaries with claims associated with fraudulent
physicians
        fraud_beneficiaries = df[
            (df.AttendingPhysician.isin(set_physicians)) &
            (df.PotentialFraud == 1)
        ]["BeneID"].unique()
        new_fraud_beneficiaries =
set(fraud_beneficiaries).difference(set_beneficiaries)
        # Upate set with new fraudulent beneficiaries
        set_beneficiaries.update(new_fraud_beneficiaries)

        # update variables
        i += 1
        current_cluster_size = len(set_physicians) + len(set_beneficiaries)

    # Final dataset of physicians and beneficiaries in sets with fraudulent
claims
    df_results = df[
        (df.AttendingPhysician.isin(set_physicians)) &
        (df.BeneID.isin(set_beneficiaries)) &
        (df.PotentialFraud == 1)
    ]

    return df_results
```

Now that you know how to get connections of fraudulent claims associated with a given physician, let's pick a random physician. Let's use AttendingPhysician `PHY379763` as an example, who has only 2 claims but both were fraudulent.

After using the function above to find this physician's connections up to 2 hops away, the resulting dataset has 1,483 fraudulent claims containing 920 unique beneficiaries and 2 other physicians (so 3 distinct physicians altogether).

It is crazy to think that looking at the connections to a physician with only 2 claims can expand so quickly!

Creating the Time Series Network Plot

After gathering the connections data, we are now ready to create a visualization. This can be broken down into 4 steps: i) building the main graph, ii) drawing a graph per time interval, iii) plotting the graphs per time interval, iv) configuring the figure.

I. Building the Graph

To begin, initialize an empty graph using `networkx`and add data to it from the DataFrame. In this example, the graph will contain two sets of nodes: one set of nodes for beneficiaries and another set for physicians.

It is important to label the nodes with {type, ID (`BeneID`or `AttendingPhysician`), claim start date (`ClaimStartDt`)}, because later on we will use the date to select nodes to remove for our monthly graphs.

Next, add the edges between connections. In this case, the edges exist between connected beneficiaries and physicians, and the label of these edges contain the {beneficiary ID (`BeneID`), physician ID (`AttendingPhysician`), claim start date (`ClaimStartDt`)}. Again, we will use the date to select edges to remove for the monthly graphs.

Below is what this section looks like:

```
# Initialize empty networkx graph
G = nx.Graph()
# Build graph from df
for row in df.itertuples():
  # Add nodes for beneficiaries
  G.add_node(row.BeneID, label=("BeneID", row.BeneID, row.ClaimStartDt))
  # Assign a different node label (for node color later on) for root Physician
  if RootPhysicianID and row.AttendingPhysician == RootPhysicianID:
      G.add_node(row.AttendingPhysician, label=("RootPhysician",
row.AttendingPhysician, row.ClaimStartDt))
  else:
      G.add_node(row.AttendingPhysician, label=("AttendingPhysician",
row.AttendingPhysician, row.ClaimStartDt))
  # Add edges
  G.add_edge(row.BeneID, row.AttendingPhysician, label=(row.BeneID,
row.AttendingPhysician, row.ClaimStartDt))

# Get positions
pos = nx.spring_layout(G)
print("Finished building graph (G).")
```

Note that the `RootPhysicianID` is the given physician ID that we started with, and we use a different label for this node so that we can give it a distinct color later on when plotting.

Also, it is important to get the positions of nodes and edges using the`nx.spring_layout()` function, so that we can supply these positions to monthly graphs in order for nodes and edges to appear consistently in the same positions over time later on in the plotting step.

II. Draw a Graph per Interval

Next, decide on what time interval and frequency you would like your visualization to show. In this case, we will be using monthly intervals, so the resulting plot will create sliders for each month in the dataset.

Start by identifying the months to loop through by grabbing the unique months in the dataset:

```
# Grab unique months in dataset
months = [
    i.to_timestamp("D") for i in pd.to_datetime(
        df["ClaimStartDt"]
    ).dt.to_period("M").sort_values().unique()
]
```

Then, loop through each month to duplicate the original graph (`G`) and create a new graph (`new_G`) that we can modify to show data for this month. Then, identify and grab nodes and edges to remove if they occur after this month based on the claim start date in the labels. Remove these nodes, edges and any self-loops:

```
# Loop through months
for month in months:
    month_str = month.strftime("%Y-%m")

    # Duplicate original graph for a new graph in current month
    new_G = G.copy()

    # Extract nodes and edges to remove
    nodes_to_remove = [
        node for node in new_G.nodes()
        if "label" in new_G.nodes[node]
        and new_G.nodes[node]["label"][-1] >= month
    ]
    edges_to_remove = [
        (u, v) for u, v, data in new_G.edges(data=True)
        if "label" in data and data["label"][-1] >= month
    ]
    # Remove self-loops
    for u, v, label in new_G.edges(data=True):
        if u == v:
            new_G.remove_edge(u, v)

    # Remove nodes and edges from new graph
    new_G.remove_edges_from(edges_to_remove)
    new_G.remove_nodes_from(nodes_to_remove)
```

Now that we have a graph created for every month, it is time to plot them and add them to a Plotly Figure.

III. Plotting the Graph

To plot a network graph using Plotly, start by selecting the positions (x- and y-coordinates) of nodes and edges from the positions selected in the previous step stored in the variable `pos`.

We will also add extra text and/ or colors for different nodes, and then add these edges and nodes to the figure. We will choose and define colors for different nodes and store this in a dictionary called

`color_mappings`. The keys correspond to the node labels and the values are the colors' hex codes. Here is a helpful website for choosing colors that go well together.

Below is the function to plot a graph:

```
def plot_graph(G, pos, fig):
    # Define colors with hex codes
    color_mappings = {
        "BeneID": "#66bfbf", # teal green
        "AttendingPhysician": "#fecea8",  # light orange
        "RootPhysician": "#f76b8a", # pinkish red
    }
    # Select nodes
    node_x = [pos[node][0] for node in G.nodes() if node in pos]
    node_y = [pos[node][1] for node in G.nodes() if node in pos]

    # Select edges
    edge_x = []
    edge_y = []
    for u, v, label in G.edges(data=True):
        if u in pos and v in pos:
            x0, y0 = pos[u]
            x1, y1 = pos[v]
            edge_x.extend([x0, x1, None])
            edge_y.extend([y0, y1, None])

    # Additional text to show on hover
    node_text = [data["label"] for node, data in G.nodes(data=True)]
    node_colors = [color_mappings.get(data["label"][0]) for node, data in
G.nodes(data=True)]

    # Add nodes and edges to plot
    fig.add_trace(
        go.Scatter(
            x=edge_x,
            y=edge_y,
            mode="lines",
            line=dict(color="gray"),
            name="Edges"
        )
    )
    fig.add_trace(
        go.Scatter(
            x=node_x,
            y=node_y,
            mode="markers",
            marker=dict(size=10, color=node_colors),
            text=node_text,
            hoverinfo="text",
            opacity=0.8,
            name="Nodes",
        )
    )

    return fig
```

The function above is useful and will be reused to plot each monthly graph as such:

```
# Plot new graph for this month and add to figure
fig = plot_graph(G=new_G, pos=pos, fig=fig)
```

IV. Configuring Figure

Up until this step, we have a main graph (`G`) containing all nodes and edges, monthly graphs (multiple `new_G`‘s) containing all nodes and edges with claim start dates up to that month, and a Plotly figure (`fig`) containing multiple traces that plots the network graphs for each month.

Next, we have to add the steps for the slider. There should be a step for each month on the slider, and we need to configure what traces to show at each step.

Remember that we have two sets of traces, one for nodes and one for edges for each month, so we will set the visibility for these traces accordingly:

```
# Configure slider
steps = []
for i, month in enumerate(months):
    idx = i * 2   # multiply by 2 because one trace is for nodes and one is for
edges
    month_str = month.strftime("%Y-%m")
    visibility = [False] * len(months) * 2   # Make all traces not visible by
default
    visibility[idx] = True   # Make trace for nodes visible
    visibility[idx + 1] = True    # Make trace for edges visible
    step = dict(
        method="update",
        label="Month: %s" % month_str,
        args=[{"visible": visibility}],
    )
    steps.append(step)

# Make first graph visible
fig.data[1].visible = True
```

After creating steps for the slider, we can update the layout of the figure to add the slider, title and other miscellaneous configurations:

```
# Update figure settings
fig.update_layout(
    title="Time Series Graph Visualization for Root PhysicianID = %s" %
RootPhysicianID,
    showlegend=False,
    hovermode="closest",
    xaxis=dict(showgrid=False, zeroline=False, showticklabels=False),
    yaxis=dict(showgrid=False, zeroline=False, showticklabels=False),
    sliders=[
        dict(
            active=0,
            steps=steps,
            pad={"t": 50}
        ),
    ],
)
```

All that is left is to show the figure and your time series network plot should appear!

Finally, use the following command if you wish to save the interactive plot as an html file:

```
plotly.offline.plot(fig, filename="./graph_viz.html")
```

Conclusion

Network visualizations are very powerful, and adding a time series component make them even more informative! Visualizing the growth of a network has many applications in various fields, ranging from understanding the origins of a disease to following the expansion of services and more.

The example we saw involved finding fraudulent claims associated with a selected physician up to two hops away, and then plotting this out over time to see how fraud for this cluster developed.

To recap, you learned how to find connections to a given entity up to a maximum number of hop and create a time series network plot by:

i) Building the main graph (`G`),

ii) Drawing a new graph for each time interval (monthly in this case),

iii) Plotting each monthly graph (multiple `new_G`'s) and adding them to the figure (`fig`),

iv) Configuring the slider for the figure.

It is fascinating to visualize how the connections between physicians and beneficiaries grew over time!

Comment below with your thoughts and learnings. What data will you try this on? What insights did you draw from creating the visualization?

- Link to this Jupyter notebook on Github

Time Series Analysis
Network
Visualization
Data
Plotly

tds

Written by Claudia Ng

364 Followers
·Writer for

Towards Data Science

Data Scientist | FinTech | Harvard MPP | Language Enthusiast