**Group and Calculate Transitions**:****

* + Group the data by page and next\_page to calculate the frequency of transitions.
  + Sort transitions by count in descending order to prioritize the most common paths.

1. **Start with the Most Common First Page**:
   * Initialize the current\_page with the page that has the highest transition frequency.
2. **Construct the Path Iteratively**:
   * Add the current\_page to the happy\_path.
   * Check the is\_exit column to determine if the current\_page is an exit page. If is\_exit == 1, append "Exit" and terminate.
3. **Validate Transitions Using** page\_referrer:
   * Ensure that the page\_referrer matches the previous\_page to validate the transition.
   * Use this condition to filter valid next pages:

python

CopyEdit

next\_pages = transitions[(transitions['page'] == current\_page) &

(group['page\_referrer'] == previous\_page)]

1. **Handle Empty Transitions**:
   * If no valid next pages exist, append "Exit" to the happy\_path and break the loop.
2. **Return the Happy Path**:
   * Convert the happy\_path list into a string with arrows (->) indicating the sequence of pages.

### # Function to calculate the happy path

def calculate\_happy\_path(group):

# Check if the group is empty

if group.empty:

return "No Data"

# Calculate transition frequencies for the journey

transitions = group.groupby(['page', 'next\_page']).size().reset\_index(name='count')

transitions = transitions.sort\_values('count', ascending=False)

# Ensure transitions are valid

if transitions.empty:

return "No Valid Transitions"

# Start constructing the happy path

happy\_path = []

current\_page = transitions.iloc[0]['page'] # Start with the most common first page

previous\_page = None

while True:

# Add the current page to the happy path

happy\_path.append(current\_page)

# Check if the current page is an exit page

is\_exit = group[group['page'] == current\_page]['is\_exit'].iloc[0]

if is\_exit == 1: # Terminate if the current page is marked as exit

happy\_path.append('Exit')

break

# Get the most common valid transition from the current page

next\_pages = transitions[(transitions['page'] == current\_page) &

(group['page\_referrer'] == previous\_page)] # Validate page\_referrer

# If no valid next pages exist, terminate the loop

if next\_pages.empty:

happy\_path.append('Exit')

break

# Update the current and previous pages

previous\_page = current\_page

current\_page = next\_pages.iloc[0]['next\_page']

return " -> ".join(happy\_path)

# Sort data by user journey

df\_sorted = df.sort\_values(['journey\_name', 'account\_num', 'exit\_time'])

# Apply the logic journey-wise

happy\_paths = df\_sorted.groupby('journey\_name').apply(calculate\_happy\_path).reset\_index(name='happy\_path')

# Display the happy paths

print("Happy Paths for Each Journey:")

print(happy\_paths)

# Ensure transition weights are represented in the graph

if len(journey\_edges) == 0:

print(f"No journey data to plot for User {user\_to\_visualize}.")

else:

import plotly.graph\_objects as go

# Create a directed graph

G = nx.DiGraph()

# Add edges with weights (transition counts)

for edge in journey\_edges:

source, target = edge[0], edge[1]

if len(edge) > 2: # Check if weight is included in edge

weight = edge[2]

else:

weight = 1 # Default weight if not specified

G.add\_edge(source, target, weight=weight)

# Debugging graph data

print(f"Nodes: {G.nodes()}")

print(f"Edges: {G.edges(data=True)}")

# Generate a layout for the graph

pos = nx.spring\_layout(G, seed=42)

# Extract edge positions for plotting

edge\_x = []

edge\_y = []

edge\_text = []

for edge in G.edges(data=True):

x0, y0 = pos[edge[0]]

x1, y1 = pos[edge[1]]

edge\_x.extend([x0, x1, None])

edge\_y.extend([y0, y1, None])

weight = edge[2]['weight']

edge\_text.append(f"{edge[0]} → {edge[1]}: {weight} visits")

# Extract node positions and details

node\_x = []

node\_y = []

node\_text = []

for node in G.nodes:

x, y = pos[node]

node\_x.append(x)

node\_y.append(y)

node\_text.append(node) # Remove "Node" from hover labels

# Create edge traces (lines with directional arrows)

edge\_trace = go.Scatter(

x=edge\_x,

y=edge\_y,

line=dict(width=0.5, color='blue'),

hoverinfo='text',

mode='lines+markers',

marker=dict(size=4, color='blue'),

text=edge\_text

)

# Create node traces

node\_trace = go.Scatter(

x=node\_x,

y=node\_y,

mode='markers+text',

text=node\_text, # Only node names

hoverinfo='text',

textposition="top center",

marker=dict(

color=['red' if 'Exit' in node else 'lightblue' for node in G.nodes()],

size=20,

line\_width=2

)

)

# Create the Plotly figure

fig = go.Figure(data=[edge\_trace, node\_trace],

layout=go.Layout(

title=f"Interactive Journey Visualization for User {user\_to\_visualize}",

titlefont\_size=16,

showlegend=False,

hovermode='closest',

margin=dict(b=0, l=0, r=0, t=40),

annotations=[dict(

showarrow=False,

text="",

xref="paper",

yref="paper"

)],

xaxis=dict(showgrid=False, zeroline=False),

yaxis=dict(showgrid=False, zeroline=False)

))

# Show the graph

fig.show()

# Print the ordered journey

ordered\_journey = " -> ".join([edge[0] for edge in journey\_edges] + [journey\_edges[-1][1]])

print(f"Ordered Journey for User {user\_to\_visualize}:")

print(ordered\_journey)