**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\_paths) == 0:

print(f"No journey data to plot.")

else:

# Iterate over each journey\_name to create graphs for each journey

for journey\_name, journey\_data in journey\_paths.itertuples(index=False):

journey\_edges = [(path.split(' -> ')[0], path.split(' -> ')[1], count)

for path, count in journey\_data['path\_counts'].items()]

if len(journey\_edges) == 0:

print(f"No journey data to plot for Journey: {journey\_name}.")

continue

# Create a directed graph

G = nx.DiGraph()

# Add edges with weights (transition counts)

for edge in journey\_edges:

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

weight = edge[2]

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

# Debugging graph data

print(f"Journey: {journey\_name}")

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

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

# Customize node colors (highlight Exit nodes in red)

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

# Generate a layout for better clarity

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

# Plot the graph

plt.figure(figsize=(14, 10))

edge\_labels = nx.get\_edge\_attributes(G, 'weight') # Retrieve weights for edges

nx.draw\_networkx\_nodes(G, pos, node\_color=node\_colors, node\_size=1000, edgecolors='black')

nx.draw\_networkx\_edges(

G, pos,

width=[G[u][v]['weight'] for u, v in G.edges()], # Adjust width based on weight

edge\_color='blue', arrowsize=20

)

nx.draw\_networkx\_labels(G, pos, font\_size=10, font\_color='black', verticalalignment='center')

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels, font\_size=8, font\_color='darkgreen')

plt.title(f"Journey Visualization with Transition Weights for Journey: {journey\_name}", fontsize=16, pad=20)

plt.tight\_layout()

plt.show()

# Print the ordered journey

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

print(f"Ordered Journey for {journey\_name}:")

print(ordered\_journey)

# Build valid paths for each journey

df['path'] = df['page\_referrer'] + ' -> ' + df['page'] + ' -> ' + df['next\_page']

# Group paths by journey\_name and calculate unique paths

journey\_paths = df.groupby('journey\_name')['path'].apply(list).reset\_index(name='all\_paths')

# Calculate unique paths and their counts for each journey

journey\_paths['unique\_paths'] = journey\_paths['all\_paths'].apply(lambda paths: list(set(paths)))

journey\_paths['path\_counts'] = journey\_paths['all\_paths'].apply(lambda paths: {path: paths.count(path) for path in set(paths)})

# Display journey paths with counts

print("Paths and Counts for Each Journey:")

print(journey\_paths)

# Example Output Structure:

# journey\_paths

# journey\_name all\_paths unique\_paths path\_counts

# Journey A [a->b, b->c, c->exit] [a->b, b->c, c->exit] {'a->b': 2, 'b->c': 1}

# Journey B [x->y, y->z, z->exit] [x->y, y->z, z->exit] {'x->y': 1, 'y->z': 1}

# Build valid paths for each journey

df['path'] = df['page\_referrer'] + ' -> ' + df['page'] + ' -> ' + df['next\_page']

# Group paths by journey\_name and calculate unique paths

journey\_paths = df.groupby('journey\_name')['path'].apply(list).reset\_index(name='all\_paths')

# Calculate unique paths and their counts for each journey

journey\_paths['unique\_paths'] = journey\_paths['all\_paths'].apply(lambda paths: list(set(paths)))

journey\_paths['path\_counts'] = journey\_paths['all\_paths'].apply(lambda paths: {path: paths.count(path) for path in set(paths)})

# Find the happy path (most frequently taken path) for each journey

journey\_paths['happy\_path'] = journey\_paths['path\_counts'].apply(lambda counts: max(counts, key=counts.get))

# Display journey paths with counts and happy paths

print("Paths and Counts for Each Journey with Happy Paths:")

print(journey\_paths[['journey\_name', 'path\_counts', 'happy\_path']])

# Plot the happy path for each journey

for \_, row in journey\_paths.iterrows():

journey\_name = row['journey\_name']

happy\_path = row['happy\_path']

# Split the happy path into nodes for plotting

nodes = happy\_path.split(' -> ')

edges = [(nodes[i], nodes[i + 1]) for i in range(len(nodes) - 1)]

# Create a directed graph for the happy path

G = nx.DiGraph()

for edge in edges:

G.add\_edge(edge[0], edge[1], weight=row['path\_counts'][happy\_path])

# Debugging graph data

print(f"Happy Path for Journey: {journey\_name}")

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

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

# Customize node colors (highlight Exit nodes in red)

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

# Generate a layout for better clarity

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

# Plot the happy path

plt.figure(figsize=(14, 10))

edge\_labels = nx.get\_edge\_attributes(G, 'weight') # Retrieve weights for edges

nx.draw\_networkx\_nodes(G, pos, node\_color=node\_colors, node\_size=1000, edgecolors='black')

nx.draw\_networkx\_edges(

G, pos,

width=3, # Fixed arrow thickness for clarity

edge\_color='blue', arrowsize=20

)

nx.draw\_networkx\_labels(G, pos, font\_size=10, font\_color='black', verticalalignment='center')

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels, font\_size=8, font\_color='darkgreen')

plt.title(f"Happy Path Visualization for Journey: {journey\_name}", fontsize=16, pad=20)

plt.tight\_layout()

plt.show()

# 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:

# 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)}")

# Customize node colors (highlight Exit nodes in red)

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

# Generate a layout for better clarity

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

# Plot the graph

plt.figure(figsize=(14, 10))

edge\_labels = nx.get\_edge\_attributes(G, 'weight') # Retrieve weights for edges

nx.draw\_networkx\_nodes(G, pos, node\_color=node\_colors, node\_size=1000, edgecolors='black')

nx.draw\_networkx\_edges(G, pos, width=2, edge\_color='blue', arrowsize=20)

nx.draw\_networkx\_labels(G, pos, font\_size=10, font\_color='black', verticalalignment='center')

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels, font\_size=8, font\_color='darkgreen')

plt.title(f"Journey Visualization with Transition Weights for User ID: {user\_to\_visualize}", fontsize=16, pad=20)

plt.tight\_layout()

plt.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)

import plotly.graph\_objects as go

# Ensure transition weights are represented in the Sankey diagram

if len(journey\_edges) == 0:

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

else:

# Create lists for Sankey Diagram

sources = []

targets = []

weights = []

labels = set()

# 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

sources.append(source)

targets.append(target)

weights.append(weight)

labels.add(source)

labels.add(target)

# Convert labels to list and create a mapping for indices

labels = list(labels)

label\_to\_index = {label: i for i, label in enumerate(labels)}

# Convert sources and targets to indices

sources\_indices = [label\_to\_index[source] for source in sources]

targets\_indices = [label\_to\_index[target] for target in targets]

# Debugging data for Sankey Diagram

print(f"Labels: {labels}")

print(f"Sources: {sources\_indices}")

print(f"Targets: {targets\_indices}")

print(f"Weights: {weights}")

# Generate labels for links (arrows) with transition weights

link\_labels = [f"{weight} transitions" for weight in weights]

# Create the Sankey Diagram

fig = go.Figure(go.Sankey(

node=dict(

pad=15,

thickness=20,

line=dict(color="black", width=0.5),

label=labels,

color=['red' if 'Exit' in label else 'lightblue' for label in labels]

),

link=dict(

source=sources\_indices,

target=targets\_indices,

value=weights,

color='blue', # Uniform link color

customdata=link\_labels,

hovertemplate="From %{source.label} to %{target.label}<br>%{value} transitions<extra></extra>",

)

))

fig.update\_layout(

title\_text=f"Sankey Diagram with Transitions for User ID: {user\_to\_visualize}",

font\_size=10

)

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)