Cafe Rewards Program: Interaction Patterns and Completion Rates

Introduction and Background

I conducted this analysis in response to a request from the marketing team, seeking insights into customer interactions with various reward offers. Below is the original email detailing the specific objectives and requirements for the analysis:


10:04 4G

Inbox

Monica
To: me

08/11/2024

Data Request: Performance Insights on Recent Rewards Offers

Hi Data Team,

The Marketing Team is preparing a presentation for the upcoming company-wide town hall and would like insights on the impact of our latest rewards program. We need an analysis to cover the following areas:

• Completion Rates: How many reward offers were completed, and which offers had the highest completion rates? Please include informational offers—our leadership has decided that any purchases made within 72 hours of viewing an informational offer should be counted as that offer's completion.

• Customer Demographics: What are the key demographic patterns, especially regarding offer completion?

Your insights will be invaluable for understanding how our customers are engaging with the rewards program and for planning future initiatives.

Thanks so much - Monica @ beaniecoffee.com

Data Dictionary

1. `offers` Table

Contains details about each offer available to customers:

- **offer_id**: A unique code for each offer.
- **offer_type**: The type of offer:
  - **BOGO**: Buy one, get one free.
  - **Discount**: Money off on purchases.
  - **Informational**: No reward, just information on products.
- **difficulty**: Minimum amount a customer must spend to qualify for the reward.
- **reward**: Dollar amount a customer earns for completing the offer (only for BOGO and Discount offers).

- **duration**: The number of days the customer has to complete the offer after receiving it.
- **channels**: Methods used to send the offer to customers (e.g., email, social media).

2. `customers` Table

Information about each customer in the rewards program:

- **customer_id**: A unique identifier for each customer.
- **became_member_on**: The date the customer joined the rewards program.
- **gender**: The customer's gender (Male, Female, Other).
- **age**: The customer's age.
- **income**: The customer's estimated annual income (in USD).

3. `events` Table

Tracks customer actions over a 30-day period, including transactions and interactions with offers:

- **customer_id**: Links this record to a specific customer.
- **event**: Describes what happened, with possible types:
  - **Transaction**: The customer made a purchase.
  - **Offer received**: The customer got an offer.
  - **Offer viewed**: The customer looked at an offer.
  - **Offer completed**: The customer met the conditions of the offer and earned the reward.
- **value**: Contains specific details for each event:
  - For **transaction** events: The amount spent.
  - For **offer-related** events (received, viewed, completed): The `offer_id`.
  - For **offer completed** events: The `offer_id` is appended with the word "reward".
- **time**: The number of hours since the start of the 30-day period when the event occurred.

How It All Connects

- **Offers and events**:
  Each time a customer receives an offer, an "offer received" event appears in the `events` table.
  - If the customer views the offer, it's logged as "offer viewed."
  - Completing the offer (meeting requirements and spending enough) generates an "offer completed" event with the reward earned.
- **Transactions and offers**:
  Transactions capture customer spending. If a transaction occurs while completing an offer, it can be **attributed** to that offer.

Setup and Imports

I'm using the following libraries:

- `pandas` : For data manipulation and analysis.
- `seaborn` and `matplotlib.pyplot` : For visualizations.
- `ast` : For evaluating and parsing string representations of Python literals (in this example dictionaries) into actual Python objects.
- `time` : For tracking script run time.
- `sqlite3` : For saving data to a sqlite3 database.

In [ ]:
```
# Import the libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import ast
import time
import sqlite3

# Load the offers and customers data
offers = pd.read_csv("source-data/offers.csv")
customers = pd.read_csv("source-data/customers.csv")

# Load and combine the events data from the two parts (due to GitHub 25 MB size
events_pt1 = pd.read_csv("source-data/events-pt1.csv")
events_pt2 = pd.read_csv("source-data/events-pt2.csv")
events = pd.concat([events_pt1, events_pt2], ignore_index=True)
```

Data Cleaning & Merging

In this section, I clean the data by standardizing columns and handling any discrepancies. First, I remove `events` related to inactive `customers` (age 118). Then, I clean up the `value` column to standardize its structure and extract the `offer_id` . I also strip whitespace from all columns to ensure consistency. Afterward, I remove duplicates across the dataframes. I also map `offer_id` values to more readable names for clarity. Finally, I merge the cleaned data from the `events` , `offers` , and `customers` dataframes into one unified dataset for further analysis.

In [ ]:
```
# Step 1: Identify customers with age 118
inactive_customers = customers[customers['age'] == 118]['customer_id']
print(f"Number of events before filtering: {len(events)}")

# Step 2: Filter out events for these customers
events = events[~events['customer_id'].isin(inactive_customers)]
print(f"Number of events after filtering: {len(events)}")

# Step 3: Clean the 'value' column to standardize the keys
events['value'] = events['value'].apply(lambda x: x.replace('offer id', 'offer_i

# Step 4: Extract and clean offer_id from the 'value' column
def extract_offer_id(value):
    try:
        # Convert string representation of dict into an actual dictionary
        value_dict = ast.literal_eval(value)
        if isinstance(value_dict, dict):
            # Clean up whitespace in dictionary keys and values
```

```
            value_dict = {k.strip(): v.strip() if isinstance(v, str) else v for
            return value_dict.get('offer_id')
        return None
    except Exception as e:
        print(f"Error parsing value: {value} - {e}")
        return None

events['offer_id'] = events['value'].apply(extract_offer_id)

# Step 5: Strip whitespace from all columns in the dataframes
def strip_whitespace(df):
    return df.apply(lambda col: col.map(lambda x: x.strip() if isinstance(x, str

offers = strip_whitespace(offers)
customers = strip_whitespace(customers)
events = strip_whitespace(events)

# Step 6: Check and drop duplicates from each dataframe
offers_duplicates = offers[offers.duplicated()]
print(f"Number of duplicates in offers: {len(offers_duplicates)}")
customers_duplicates = customers[customers.duplicated()]
print(f"Number of duplicates in customers: {len(customers_duplicates)}")
events_duplicates = events[events.duplicated()]
print(f"Number of duplicates in events: {len(events_duplicates)}")

offers = offers.drop_duplicates()
customers = customers.drop_duplicates()
events = events.drop_duplicates()

# Step 7: Print total rows in events DataFrame
total_rows = events.shape[0]
print(f"Total rows in events DataFrame: {total_rows}")

# Step 8: Analyze inactive customers (age 118)
inactive_customers_count = customers[customers['age'] == 118].shape[0]
total_customers_count = customers.shape[0]
inactive_percentage = (inactive_customers_count / total_customers_count) * 100
print(f"Inactive customers (age 118): {inactive_customers_count}")
print(f"Percentage of inactive customers: {inactive_percentage:.2f}%")

# Step 9: Exclude inactive customers from the dataset
customers = customers[customers['age'] != 118]

# Step 10: Map offer IDs to names
offer_names = {
    'ae264e3637204a6fb9bb56bc8210ddfd': 'bogo_1',
    '4d5c57ea9a6940dd891ad53e9dbe8da0': 'bogo_2',
    '9b98b8c7a33c4b65b9aebfe6a799e6d9': 'bogo_3',
    'f19421c1d4aa40978ebb69ca19b0e20d': 'bogo_4',
    '0b1e1539f2cc45b7b9fa7c272da2e1d7': 'discount_1',
    '2298d6c36e964ae4a3e7e9706d1fb8c2': 'discount_2',
    'fafdcd668e3743c1bb461111dcafc2a4': 'discount_3',
    '2906b810c7d4411798c6938adc9daaa5': 'discount_4',
    '3f207df678b143eea3cee63160fa8bed': 'informational_1',
    '5a8bc65990b245e5a138643cd4eb9837': 'informational_2'
}
offers['offer_name'] = offers['offer_id'].map(offer_names)

# Step 11: Create and connect to a 'cafe.db' database
conn = sqlite3.connect("db/cafe.db")
```

```
c = conn.cursor()

# Write the 'offers' df to the database
offers.to_sql("offers", conn, if_exists="replace", index=False)

# Step 12: Merge dataframes into merged_data
merged_data = events.merge(offers, on='offer_id', how='left').merge(customers, c
total_rows = merged_data.shape[0]
print(f"Total rows in merged_data DataFrame: {total_rows}")
```

```
Number of events before filtering: 306534
Number of events after filtering: 272762
Number of duplicates in offers: 0
Number of duplicates in customers: 0
Number of duplicates in events: 374
Total rows in events DataFrame: 272388
Inactive customers (age 118): 2175
Percentage of inactive customers: 12.79%
Total rows in merged_data DataFrame: 272388
```

Calculation of Informational Offers Completions

In this step, I identify and exclude transactions that are either part of other offers or unrelated to any promotional efforts. I set up a 72-hour window in which a customer must view an informational offer and make a transaction (of any amount) for the offer to be considered completed. Only transactions that meet these criteria are included in the analysis.

- First, I exclude transactions that are immediately followed by an 'offer completed' event at the same time. This step ensures that transactions related to the completion of other offers are not counted.
- Next, I consider transactions that occur within the 72-hour window after a 'viewed' event for an informational offer. Only these transactions are included in the final list.
- I compile the filtered transactions into a list and convert it into a DataFrame for further analysis.

In [17]:
```
# Record the start time
start_time = time.time()

# Step 1: Sort merged_data by customer_id and time to ensure events are in seque
merged_data_sorted = merged_data.sort_values(by=['customer_id', 'time']).reset_i

# Number of rows in the dataset
num_rows = len(merged_data_sorted)

# Initialize a set to store indices of transactions to exclude
excluded_transactions = set()

# Exclude transactions immediately followed by 'offer completed' at the exact sa
for idx, row in merged_data_sorted.iterrows():
    if row['event'] == 'transaction' and idx + 1 < len(merged_data_sorted):
        # Check the next row for an 'offer completed' event with the same custom
        next_row = merged_data_sorted.iloc[idx + 1]
        if next_row['event'] == 'offer completed' and next_row['customer_id'] ==
            excluded_transactions.add(idx)
```

```python
# Step 2: Include only relevant transactions for informational offers within a 7
informational_completions = 0
cutoff_hours = 72  # 72-hour window

# Initialize a list to collect rows that qualify as informational completions
informational_completions_list = []

# List of offer IDs that are informational
informational_offer_ids = [key for key, value in offer_names.items() if "informa

for idx, row in merged_data_sorted.iterrows():
    if idx in excluded_transactions:
        continue

    # Check for 'offer viewed' event related to any informational offers
    if row['event'] == 'offer viewed' and row['offer_id'] in informational_offer
        view_time = row['time']
        customer_id = row['customer_id']
        offer_id = row['offer_id']  # Get the specific offer_id directly from th

        # Find all transactions for the same customer within the 72-hour cutoff
        qualifying_transactions = merged_data_sorted[
            (merged_data_sorted['customer_id'] == customer_id) &
            (merged_data_sorted['event'] == 'transaction') &
            (merged_data_sorted['time'] > view_time) &
            (merged_data_sorted['time'] <= view_time + cutoff_hours)
        ]

        # Append each qualifying transaction to the list, tagging it with the re
        for _, transaction_row in qualifying_transactions.iterrows():
            transaction_entry = transaction_row.copy()  # Copy the row to avoid
            transaction_entry['offer_id'] = offer_id  # Assign the specific info
            informational_completions_list.append(transaction_entry)

# Convert the list to a DataFrame
informational_completions_df = pd.DataFrame(informational_completions_list)

# Record the end time
end_time = time.time()

# Print the running stats
print(f"Run time: {(end_time - start_time)/60:.2f} minutes")
print(f"Number of rows processed: {num_rows}")

# Display the first three rows of informational_completions_df
print("Informational completions:")
informational_completions_df.head(3)
```

```
Run time: 7.81 minutes
Number of rows processed: 272388
Informational completions:
```

Out[17]:

	customer_id	event	value	time	
2	0009655768c64bdeb2e877511632db8f	transaction	{'amount': 22.16}	228	5a8bc65990b245e5a
6	0009655768c64bdeb2e877511632db8f	transaction	{'amount': 8.57}	414	3f207df678b143eea
55	0020c2b971eb4e9188eac86d93036a77	transaction	{'amount': 27.94}	696	5a8bc65990b245e5a

Summing Up Interaction Counts by Offer Type

In this section, I create columns in the `customers` DataFrame to count each level of interaction (received, viewed, and completed) for each specific offer type. It's essential for understanding customer engagement across different offers and supporting demographic analyses of customer behavior.

- **Received**: Counts how many times each customer received an offer.
- **Viewed**: Counts how many times each customer viewed the offer, following up on the initial receipt.
- **Completed**: Counts offer completions. For informational offers, this is defined as a transaction occurring within a 72-hour window after viewing the offer. For all other offers, this refers to explicit "offer completed" events.

The process iterates through each offer type and populates these counts in the `customers` DataFrame:

1. **Initialize Columns**: First, I set up columns for each interaction level (`received` , `viewed` , `completed`) for all offer types in `offer_names` .
2. **Filter by Offer Type**: For each `offer_id` :
   - I filter the events data to focus only on this specific offer.
3. **Calculate Counts**:
   - **Received**: Counts 'offer received' events per customer.
   - **Viewed**: Counts 'offer viewed' events per customer.
   - **Completed (Informational Offers)**: For informational offers, I track completions based on transactions within 72 hours after viewing.
   - **Completed (Other Offers)**: For other offers, I count 'offer completed' events directly.
4. **Update Customers DataFrame**: Each count is then updated in the respective columns for the corresponding customers, allowing for easy aggregation, comparison, and demographic breakdowns.

In [18]:
```
# Record the start time
start_time = time.time()

# Number of rows in the dataset
num_rows = len(customers)

# Initialize columns in the customers DataFrame for each offer event (received,
```

```
for offer_name in offer_names.values():
    customers[f"{offer_name}_received"] = 0
    customers[f"{offer_name}_viewed"] = 0
    customers[f"{offer_name}_completed"] = 0

# Iterate over each offer type and populate counts in the customers DataFrame
for offer_id, offer_name in offer_names.items():
    # Filter events for this specific offer_id
    offer_events = merged_data[merged_data['offer_id'] == offer_id]

    # Count 'received' events for each customer and update in customers DataFram
    received_counts = offer_events[offer_events['event'] == 'offer received']['c
    for customer_id, count in received_counts.items():
        customers.loc[customers['customer_id'] == customer_id, f"{offer_name}_re

    # Count 'viewed' events for each customer and update in customers DataFrame
    viewed_counts = offer_events[offer_events['event'] == 'offer viewed']['custo
    for customer_id, count in viewed_counts.items():
        customers.loc[customers['customer_id'] == customer_id, f"{offer_name}_vi

    # For informational offers, count transactions that happened within 72 hours
    if 'informational' in offer_name:
        # Get the informational offer_id
        informational_completions = informational_completions_df[informational_c

        # Count the number of completed transactions for each customer related t
        completed_counts = informational_completions['customer_id'].value_counts
        for customer_id, count in completed_counts.items():
            customers.loc[customers['customer_id'] == customer_id, f"{offer_name

    # Count 'completed' events for each customer and update in customers DataFra
    if 'informational' not in offer_name:
        completed_counts = offer_events[offer_events['event'] == 'offer complete
        for customer_id, count in completed_counts.items():
            customers.loc[customers['customer_id'] == customer_id, f"{offer_name

# Record the end time
end_time = time.time()

# Print the running stats
print(f"Run time: {(end_time - start_time)/60:.2f} minutes")
print(f"Number of rows processed: {num_rows}")

# Display the first three rows of the "customers" dataframe
print("'Customers' dataframe:")
customers.head(3)
```

```
Run time: 4.14 minutes
Number of rows processed: 14825
'Customers' dataframe:
```

Out[18]:

	customer_id	became_member_on	gender	age	income	bog
1	0610b486422d4921ae7d2bf64640c50b	20170715	F	25	112000.0	
3	78afa995795e4d85b5d9ceeca43f5fef	20170509	F	45	100000.0	
5	e2127556f4f64592b11af22de27a7932	20180426	M	38	70000.0	

3 rows × 35 columns

Summing Up Offer Interactions and Calculating Completion Rates

In this part I summarize interactions with each offer type by detailing the number of times each offer was `received`, `viewed`, and `completed`, and calculating the completion rate for each offer. This summary helps analyze the effectiveness of each offer type and evaluate its success rate and level of engagement.

Detailed Steps:

1. **Identify Offer Columns**:
   - I find all columns in the `customers` DataFrame that track the `received` interactions for each offer. These columns are used to calculate the total interactions for each offer type.
2. **Initialize Data Dictionary**:
   - I prepare a dictionary, `offer_data`, to store the aggregated data for each offer type. The dictionary contains keys for `offer_received`, `offer_viewed`, `offer_completed`, and `completion_rate`.
3. **Summing Values and Calculating Completion Rate**:
   - For each offer type:
     - I extract the base name of the offer (e.g., `bogo_1`) by splitting the column name.
     - I identify the related columns for `viewed` and `completed` events.
     - I calculate the total counts of `received`, `viewed`, and `completed` interactions for each offer by summing the values in the respective columns.
     - I calculate the **completion rate** as:
       completion rate = (total completed / total received) * 100

       If there were no `received` interactions, I set the completion rate to 0 to avoid division by zero.
4. **Create a DataFrame for the Summary**:
   - I convert `offer_data` into a DataFrame, `offers_summary`, with the base offer names as the index. This table provides a clear overview of the

engagement levels for each offer, useful for reporting and demographic analysis.

5. **Display the Result**:
    - I print the `offers_summary` DataFrame to display the completion rates and counts of each interaction type by offer.

In [19]:
```
# Step 1: Identify columns in the 'customers' dataframe related to offers receiv
offer_columns = [col for col in customers.columns if '_received' in col]

# Step 2: Initialize a dictionary to hold the data for the table
offer_data = {
    'offer_received': [],
    'offer_viewed': [],
    'offer_completed': [],
    'completion_rate': []
}

# Step 3: Sum the values for each offer and calculate completion rates
for offer in offer_columns:
    offer_name = offer.split('_')[0] + '_' + offer.split('_')[1]  # Extract base

    # Get the corresponding '_viewed' and '_completed' columns
    viewed_col = offer.replace('_received', '_viewed')
    completed_col = offer.replace('_received', '_completed')

    # Sum up the values for each of the three columns
    offer_data['offer_received'].append(customers[offer].sum())
    offer_data['offer_viewed'].append(customers[viewed_col].sum())
    offer_data['offer_completed'].append(customers[completed_col].sum())

    # Calculate the completion rate
    received = customers[offer].sum()
    completed = customers[completed_col].sum()
    completion_rate = (completed / received) * 100 if received != 0 else 0
    offer_data['completion_rate'].append(completion_rate)

# Step 4: Create a DataFrame from the collected data
offers_summary = pd.DataFrame(offer_data, index=[offer.split('_')[0] + '_' + off

# Step 5: Print the result
print("Detailed Completion Rates by Offer:")
print(offers_summary)
```

```
Detailed Completion Rates by Offer:
                offer_received  offer_viewed  offer_completed  \
bogo_1                    6682          5900             3603
bogo_2                    6593          6329             3281
bogo_3                    6684          3498             4141
bogo_4                    6576          6310             4074
discount_1                6726          2215             3306
discount_2                6655          6379             4847
discount_3                6651          6406             4956
discount_4                6630          3459             3859
informational_1           6657          3487             3328
informational_2           6642          5872             6172

                completion_rate
bogo_1                53.920982
bogo_2                49.764902
bogo_3                61.953920
bogo_4                61.952555
discount_1            49.152542
discount_2            72.832457
discount_3            74.515111
discount_4            58.205128
informational_1       49.992489
informational_2       92.923818
```

Aggregating Offer Interaction Counts by Offer Type

In this step, I aggregate the counts of interactions (received, viewed, and completed) for each offer type (e.g., "bogo", "discount", "informational"). I also calculate the overall completion rate for each offer group (offer type), based on how many times offers were received, viewed, and completed.

Detailed Steps:

1. **Mapping Offer Types**:
   - I define a dictionary `offer_type_mapping` to map each offer type (e.g., "bogo", "discount", "informational") to its respective offer columns (e.g., `bogo_1`, `bogo_2`, etc.).
2. **Aggregating Counts**:
   - For each offer type, I loop through the specific offers (e.g., `bogo_1`, `bogo_2`, etc.) and aggregate:
     - The total number of `received` interactions.
     - The total number of `viewed` interactions.
     - The total number of `completed` transactions for each offer type.
3. **Completion Rate Calculation**:
   - I calculate the completion rate for each offer type as:

   ```
   completion rate = (completed / received) * 100
   ```

   This gives the percentage of completed offers out of the total received ones. You can switch this to a rate based on `viewed` if desired, by modifying the formula.
4. **Displaying Aggregated Data**:

- I store the aggregated data for each offer type in a DataFrame `aggregated_df`, which shows the total `received`, `viewed`, `completed` counts, and the corresponding `completion_rate` for each offer group.

5. **Total Summary**:

   - I create a summary DataFrame called `total_summary` that aggregates the total counts of `received`, `viewed`, and `completed` for all offers combined, along with the overall completion rate.

In [20]:
```
# Define columns in customers data and map offer types
offer_type_mapping = {
    'bogo': ['bogo_1', 'bogo_2', 'bogo_3', 'bogo_4'],
    'discount': ['discount_1', 'discount_2', 'discount_3', 'discount_4'],
    'informational': ['informational_1', 'informational_2']
}

# Aggregate counts by offer type
aggregated_data = {}
for offer_type, offers in offer_type_mapping.items():
    received_columns = [f"{offer}_received" for offer in offers]
    viewed_columns = [f"{offer}_viewed" for offer in offers]
    completed_columns = [f"{offer}_completed" for offer in offers]

    aggregated_data[offer_type] = {
        'received': customers[received_columns].sum().sum(),
        'viewed': customers[viewed_columns].sum().sum(),
        'completed': customers[completed_columns].sum().sum()
    }

# Create DataFrame from aggregated data
aggregated_df = pd.DataFrame(aggregated_data).T
aggregated_df['completion_rate'] = (aggregated_df['completed'] / aggregated_df['
# If you'd like to change the calculation logic of completion rate to viewed vs
# aggregated_df['completion_rate'] = (aggregated_df['completed'] / aggregated_df
aggregated_df['completion_rate'] = aggregated_df['completion_rate'].round(0)

# Display the aggregated data
print("\nCompletion Rates by Offer Group")
print(aggregated_df)

# Calculating totals
total_summary = pd.DataFrame({
    'offer_received': [offers_summary['offer_received'].sum()],
    'offer_viewed': [offers_summary['offer_viewed'].sum()],
    'offer_completed': [offers_summary['offer_completed'].sum()]
})

# Calculate the overall completion rate
total_summary['completion_rate'] = (total_summary['offer_completed'] / total_sum

# Display the summary
print("\nTotal Summary of Completion Rates across All Offers:")
print(total_summary)
```

```
Completion Rates by Offer Group
               received  viewed  completed  completion_rate
bogo              26535   22037      15099             57.0
discount          26662   18459      16968             64.0
informational     13299    9359       9500             71.0

Total Summary of Completion Rates across All Offers:
   offer_received  offer_viewed  offer_completed  completion_rate
0           66496         49855            41567        62.510527
```

Visualizing Offer Interactions with a Heatmap

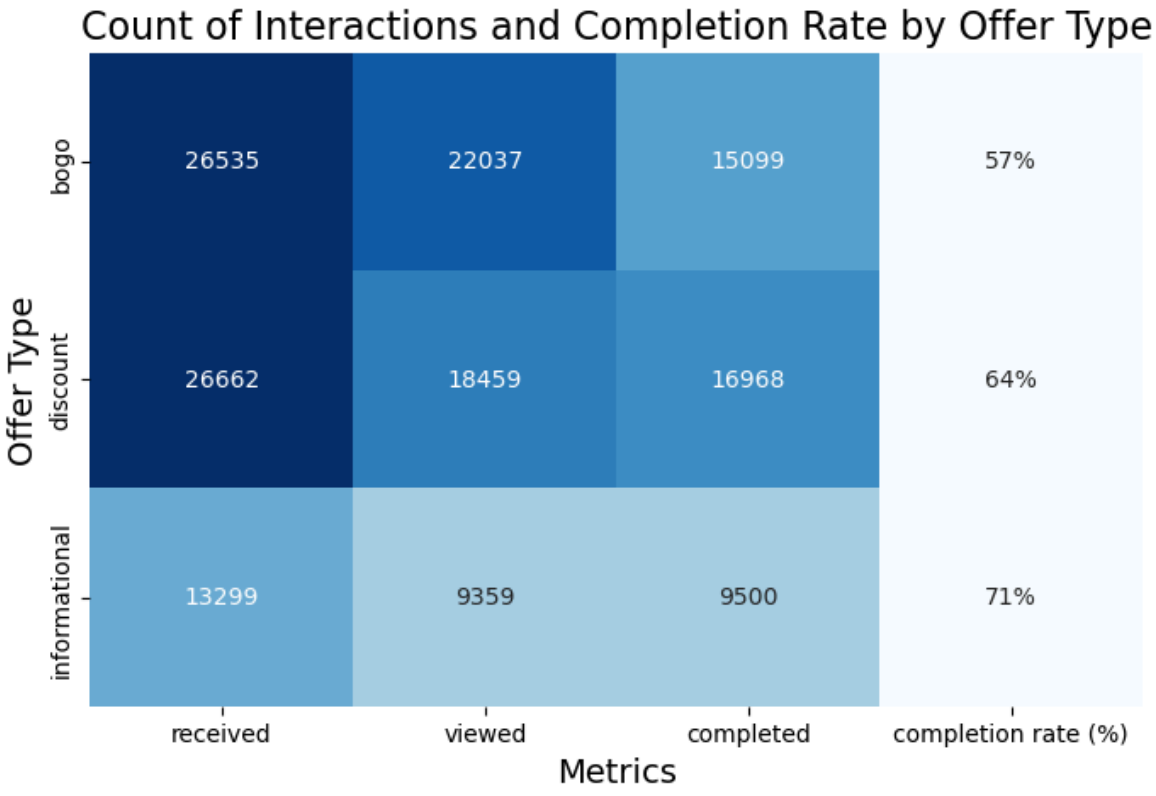
In this section, I create a heatmap to visualize the aggregated interactions for each offer type across metrics such as "received," "viewed," "completed," and the completion rate. The heatmap provides a compact way to observe the overall performance of each offer type in terms of user interactions.

Steps:

1. **Updating Column Labels**:
   - I update the column labels to provide clearer titles for the interaction metrics: "received," "viewed," "completed," and "completion rate (%)."
2. **Formatting Annotations**:
   - I create a custom function, `custom_annotate`, to format the values in the heatmap:
     - The "completion rate" values are annotated with a percentage sign ("%").
     - The other values are displayed as integers.
3. **Generating the Heatmap**:
   - Using `seaborn`, I generate the heatmap from the `aggregated_df` DataFrame.
   - I set the `annot` argument to the `annotated_df` DataFrame to display the formatted values directly on the heatmap.
   - The color scheme, `Blues`, highlights the intensity of the values.
4. **Customizing the Axes**:
   - The x-axis represents the interaction types (e.g., "received," "viewed").
   - The y-axis represents the offer types (e.g., "bogo," "discount").
5. **Displaying the Heatmap**:
   - I use `matplotlib` to display the heatmap, ensuring the axis labels are clear and the visualization is easy to interpret.

In [21]:
```
# Define the new label for completion_rate
column_labels = aggregated_df.columns.tolist()
column_labels = ["received", "viewed", "completed", "completion rate (%)"]  # Aa

# Define a custom annotation function
def custom_annotate(data):
    # Check if the current row is 'completion_rate' to add % sign
```

```
    return data.apply(lambda x: f"{int(x)}%" if data.name == 'completion_rate' e

# Create a formatted DataFrame for annotations
annotated_df = aggregated_df.apply(custom_annotate)

# Draw the heatmap with custom annotation
plt.figure(figsize=(8, 5))

ax = sns.heatmap(
    aggregated_df,
    annot=annotated_df,
    fmt="",
    cmap="Blues",
    cbar=False
)

# Set custom labels for the columns
ax.set_xticklabels(column_labels)

# Set x and y axis labels (titles)
plt.title('Count of Interactions and Completion Rate by Offer Type', fontsize=16
plt.ylabel('Offer Type', fontsize=14)
plt.xlabel('Metrics', fontsize=14)

# Display plot
plt.show()
```



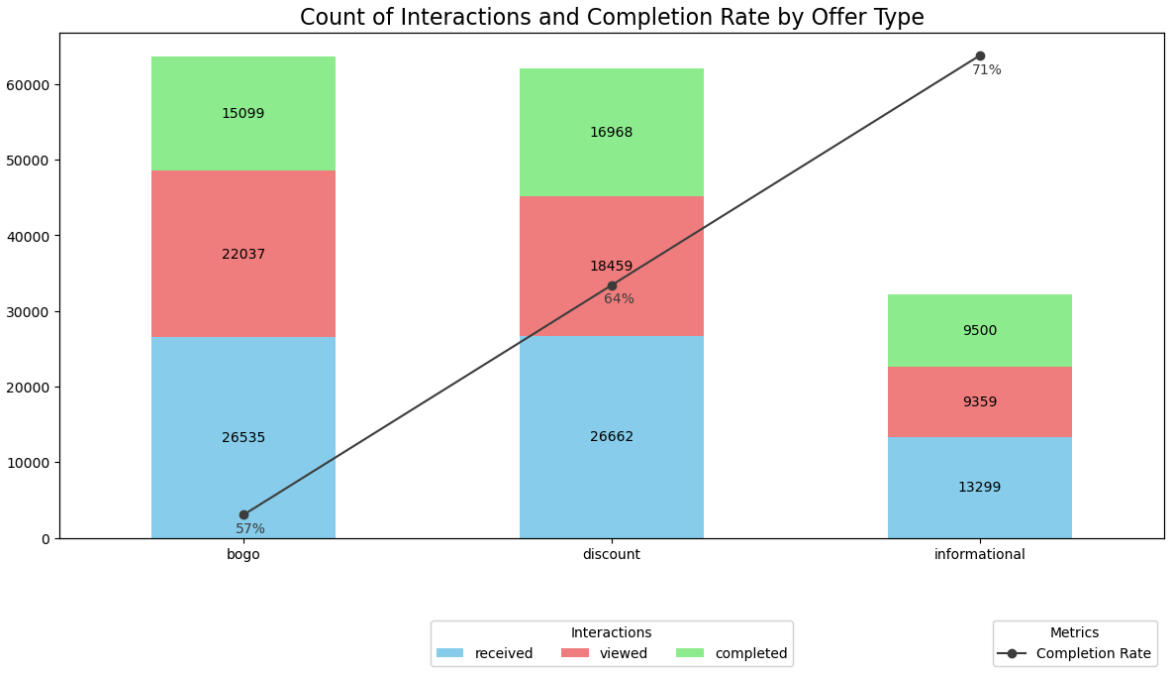
Alternative Chart Visual

This chart visually represents **offer interactions** (received, viewed, and completed) for each offer type, as well as the **completion rate** for those offers. It combines stacked bar segments and a line plot, providing a clear overview of both the interaction counts and associated completion rates.

1. **Bar Chart**:
    - I use a **stacked bar chart** to show the total counts of interactions for each offer type.
    - The bars are divided into three segments:
        - **Received** (sky blue)
        - **Viewed** (light coral)
        - **Completed** (light green)
    - The **height of each segment** represents the total count for that interaction type (received, viewed, or completed) for each offer.
    - **Labels** with counts are placed at the center of each bar segment to provide exact values for each interaction type.
2. **Completion Rate Line**:
    - I plot a **completion rate line** using a secondary y-axis, overlaid on the bar chart.
    - The line represents the percentage of **offers completed** relative to **offers received**.
    - I annotate the completion rate along the line with rounded percentages for each offer type.
3. **Legends**:
    - I split the legend into two parts:
        - **Interactions**: The legend for the stacked bar chart is located at the top center and includes categories for the received, viewed, and completed interactions.
        - **Metrics**: The legend for the completion rate line is positioned at the top right and indicates the line's color (gray).
4. **Labels and Axes**:
    - The **X-axis** represents the different **offer types** (e.g., bogo, discount, informational).
    - The **primary y-axis** shows the **count** of interactions (received, viewed, and completed).
    - The **secondary y-axis** is used for the **completion rate**, expressed as a percentage.

In [22]:
```
plt.figure(figsize=(12, 7))

# Plot the stacked bar chart for received, viewed, and completed counts
ax = aggregated_df[['received', 'viewed', 'completed']].plot(
    kind='bar', stacked=True, color=['skyblue', 'lightcoral', 'lightgreen'], ax=
)

# Add labels with counts on each bar segment
for p in ax.patches:
    height = p.get_height()
    width = p.get_width()
    x, y = p.get_xy()
    ax.annotate(f'{int(height)}', (x + width / 2, y + height / 2), ha='center',
```

```python
# Plot the completion rate line on the secondary y-axis
ax2 = ax.twinx()
ax2.plot(aggregated_df.index, aggregated_df['completion_rate'], color='#404040',

# Annotate the completion rate percentages without decimals
for i, val in enumerate(aggregated_df['completion_rate']):
    ax2.annotate(f'{int(val)}%',
                 (i, val),
                 textcoords="offset points",
                 xytext=(5, -13),  # Move labels below the dot
                 ha='center',
                 color='#404040')


# Hide the y-axis label and ticks for the completion rate line
ax2.get_yaxis().set_visible(False)

# Adjust the legends
# Move "Interactions" legend to the center and "Metrics" to the right, both slig
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.15), title="Interactions",
ax2.legend(loc='upper right', bbox_to_anchor=(1, -0.15), title="Metrics")

# Labels and title
plt.title('Count of Interactions and Completion Rate by Offer Type', fontsize=16
plt.xlabel('Offer Type', fontsize=14)
plt.ylabel('Count', fontsize=14)

# Set x-axis labels to be horizontal
ax.set_xticklabels(ax.get_xticklabels(), rotation=0)

# Display plot
plt.tight_layout()
plt.show()
```



Demographic Breakdown of Offer Interactions

This section of the analysis focuses on the **demographic breakdown** of offer interactions (received, viewed, and completed) by age group for each offer type (e.g., **bogo**, **discount**, and **informational**). It provides insights into how different age groups interact with the offers.

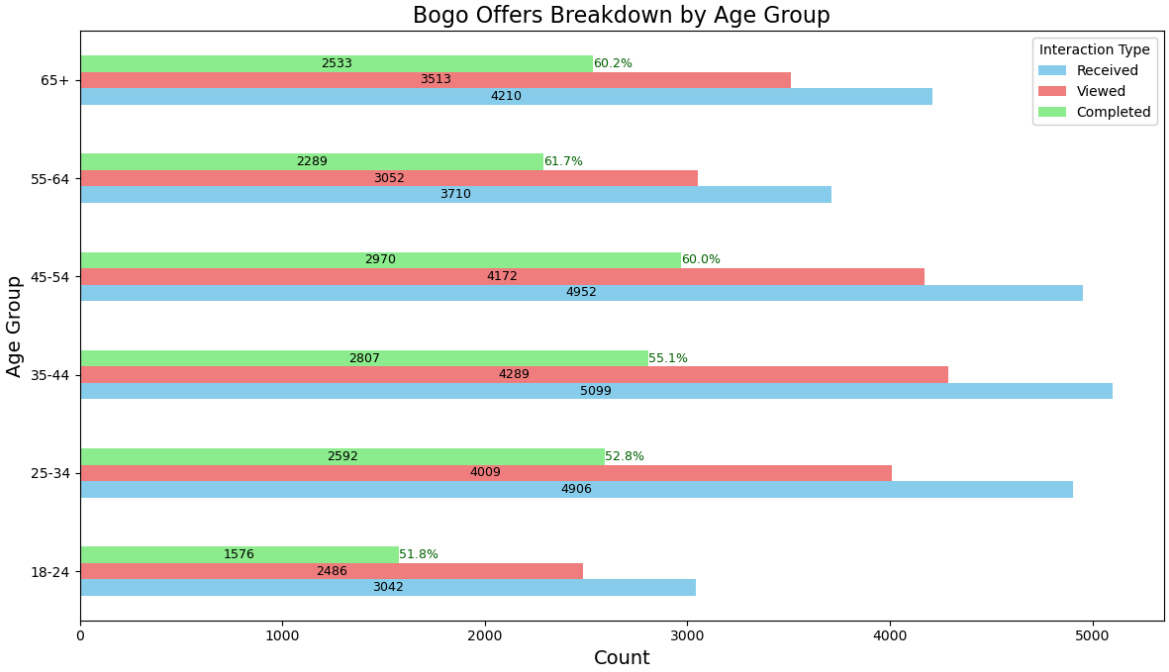
Steps:

1. **Age Grouping**:
   - I create the `age_group` column by categorizing customers into specific **age bins** (18-24, 25-34, etc.). This classification allows me to analyze how different age groups interact with offers.
2. **Data Processing**:
   - For each **offer type**, I identify relevant columns for **received**, **viewed**, and **completed** interactions by searching for columns containing both the specific offer type and interaction type in their names.
   - I perform **data grouping** by the `age_group` column, aggregating interactions for each offer type across all age groups.
3. **Visualization**:
   - I create a **horizontal bar chart** for each offer type, where the **age group** is displayed on the y-axis and the **interaction counts** (received, viewed, completed) are displayed on the x-axis.
   - Each bar represents the **count of interactions** for a particular **age group** and **offer type**. The bars are color-coded:
     - **Received** interactions: sky blue
     - **Viewed** interactions: light coral
     - **Completed** interactions: light green
   - I add **labels** with the counts to each bar segment, as well as the % completed metric.
4. **Titles and Labels**:
   - I title the chart using the offer type (e.g., "Bogo Offers Breakdown by Age Group"), giving an understanding of which offer type is being analyzed.
   - The **x-axis** represents the **count** of interactions, and the **y-axis** shows the **age group**.

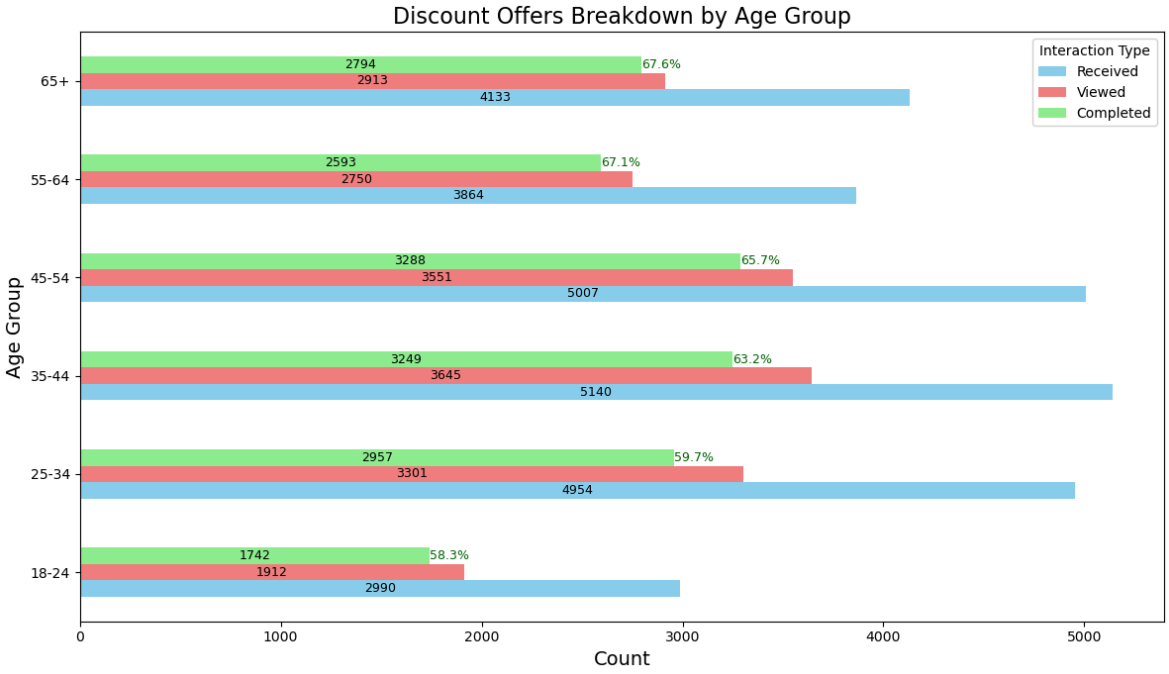
In [23]:
```
# Create age bins
bins = [18, 24, 34, 44, 54, 64, 100]
labels = ['18-24', '25-34', '35-44', '45-54', '55-64', '65+']
customers['age_group'] = pd.cut(customers['age'], bins=bins, labels=labels, righ

# Helper function to create the demographic breakdown chart
def plot_demographic_breakdown(offer_type):
    # Identify relevant columns for the current offer type (received, viewed, co
    received_columns = [col for col in customers.columns if offer_type in col an
    viewed_columns = [col for col in customers.columns if offer_type in col and
    completed_columns = [col for col in customers.columns if offer_type in col a

    # Group by age and sum the counts for each offer type (received, viewed, com
```

```
    received_by_age = customers.groupby('age_group', observed=False)[received_co
    viewed_by_age = customers.groupby('age_group', observed=False)[viewed_column
    completed_by_age = customers.groupby('age_group', observed=False)[completed_

    # Calculate the completion rate for each age group
    completion_rate = (completed_by_age / received_by_age * 100).fillna(0)

    # Combine the data into a DataFrame for plotting
    demographic_df = pd.DataFrame({
        'Received': received_by_age,
        'Viewed': viewed_by_age,
        'Completed': completed_by_age
    })

    # Plotting the data
    plt.figure(figsize=(12, 7))
    ax = demographic_df[['Received', 'Viewed', 'Completed']].plot(
        kind='barh',
        stacked=False,
        color=['skyblue', 'lightcoral', 'lightgreen'],
        figsize=(12, 7)
    )

    # Annotate count labels for all bars
    for bar in ax.patches:
        width = bar.get_width()
        height = bar.get_height()
        x, y = bar.get_xy()
        # Add the count value inside each bar
        plt.text(x + width / 2, y + height / 2, f'{int(width)}', ha='center', va

    # Annotate percentages (completion rate) next to the "Completed" bars
    for idx, bar in enumerate(ax.patches[-len(completed_by_age):]):  # Only anno
        x = bar.get_x() + bar.get_width()
        y = bar.get_y() + bar.get_height() / 2
        plt.text(x + 0.5, y, f'{completion_rate.iloc[idx]:.1f}%', va='center', f

    # Add labels and title
    plt.title(f'{offer_type.capitalize()} Offers Breakdown by Age Group', fontsi
    plt.xlabel('Count', fontsize=14)
    plt.ylabel('Age Group', fontsize=14)
    plt.legend(title='Interaction Type')
    plt.tight_layout()
    plt.show()

# Loop through the offer types and create the breakdown for each one
for offer_type in offer_type_mapping:
    plot_demographic_breakdown(offer_type)
```

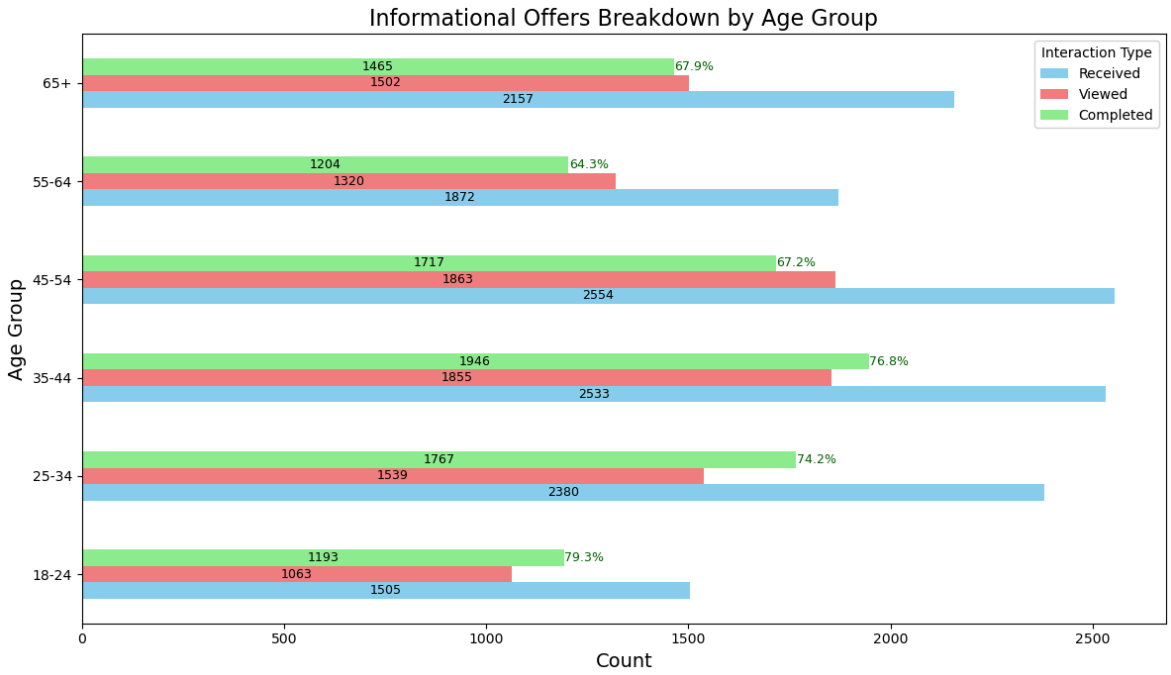
```
<Figure size 1200x700 with 0 Axes>
```



<Figure size 1200x700 with 0 Axes>



<Figure size 1200x700 with 0 Axes>



Saving Cleaned and Aggregated Data to a Database

I save the cleaned and processed data into an SQLite database using pandas **to_sql** method. SQLite is a lightweight, file-based database that suits this project because it's simple to use and sufficient for handling smaller datasets.

Here's what I do:

1. I establish a connection to the SQLite database file (cafe.db) using Python's sqlite3 library.
2. I use the to_sql method to write each DataFrame into the database, specifying:
    - The table name (e.g., offers, customers).
    - The mode of operation (if_exists="replace" ensures the table is overwritten if it already exists).
    - Exclusion of the index column from the database table (index=False).

By saving processed data to a database, I will be able to use it for future analysis and visualizations without needing to reprocess the original data repeatedly.

In [ ]:
```
customers.to_sql("customers", conn, if_exists="replace", index=False)
events.to_sql("events", conn, if_exists="replace", index=False)
informational_completions_df.to_sql("informational_completions", conn, if_exists
aggregated_df.to_sql("aggregates", conn, if_exists="replace", index=False)

# Close cursor and connection
c.close()
conn.close()
```

```
Offers Table Schema:
[(0, 'offer_id', 'TEXT', 0, None, 0), (1, 'offer_type', 'TEXT', 0, None, 0), (2,
'difficulty', 'INTEGER', 0, None, 0), (3, 'reward', 'INTEGER', 0, None, 0), (4,
'duration', 'INTEGER', 0, None, 0), (5, 'channels', 'TEXT', 0, None, 0), (6, 'off
er_name', 'TEXT', 0, None, 0)]
```

Executive Summary

Key findings include:

- A campaign-wide completion rate of ~63%.
- Informational offers showed both extreme successes (93% completion for one campaign) and weaknesses (50% completion for another).
- The success of informational offers suggests significant customer interest, but the lack of transaction-level data makes it difficult to gauge their true impact.
- Demographic trends reveal younger customers are least engaged overall, while older customers show high engagement with discount and BOGO offers.
- Older customers are consistently more engaged across most offer types.
- Younger customers engage more with informational offers but represent the smallest proportion of customers.

Completion Rates

- **Total Interactions**: 66,496 offers were received, and 41,567 were completed (~63% completion rate).
- **Informational Offers**:
  - Informational_2 excelled with a 93% completion rate.
  - Informational_1 performed poorly, with a 50% completion rate.
  - Informational offers have a short 3-day window for completion, therefore their long-term impact on purchases can't be reliably measured. Customers may interact with these campaigns beyond the 3-day window.
- **Discount Offers**:
  - Discount_3 achieved the highest success at 75% completion.
  - Discount_1 was weaker at 49%.
- **BOGO Offers**:
  - These had the overall lowest completion rate (57%), with Bogo_2 standing out as particularly weak (50%).

Customer Demographics

1. **BOGO Offers**:
   - Weakest engagement in the 18–34 age group.
   - Best performance with customers aged 55–64.
   - Most completions came from the 45–54 age group.
2. **Discount Offers**:
   - Lowest engagement from customers under 34.
   - High and consistent redemption rates from customers aged 45 and older.
3. **Informational Offers**:
   - Highest completion rates in customers aged 18–34.
   - The 45–54 age group were the biggest group who interacted with this offer.

Recommendations

1. **Informational Offers**:
    - Consider capturing transaction-level data to verify their true impact.
    - Investigate why Informational_2 was so successful and replicate its elements in future campaigns.
2. **BOGO Offers**:
    - Reassess the products included in these campaigns. They may not align with customer preferences.
    - Test alternative product selections or incentives on a small scale, borrowing successful elements from discount and informational campaigns.
    - Examine other factors such as redemption time limits, minimum spend thresholds, and marketing channels.
3. **Demographics**:
    - Increase outreach to the 18–34 demographic, who represent the smallest customer base but are responsive to informational offers.
    - Incorporate products and patterns from successful informational campaigns into BOGO and discount offers.
    - Explore whether income levels or other demographic traits influence engagement and how.