A and B Testing

April 11, 2024

1 A/B Testing using Python

What is A/B Testing? A/B testing, also known as split testing, is a method used to compare two versions of a webpage, app feature, marketing campaign, or any other aspect of a product or service. It involves dividing users or participants into two groups (A and B), where each group is exposed to a different version (A or B) of the item being tested. The objective is to determine which version performs better based on predefined metrics such as click-through rates, conversion rates, or user engagement.

1.1 Importance and Applications of A/B Testing:

1. Importance:

Data-Driven Decision Making: A/B testing allows businesses to make decisions based on empirical evidence rather than intuition or assumptions.

Optimization: It helps optimize products, services, and marketing strategies by identifying what resonates best with users or customers.

Cost-Effectiveness: By testing variations before full implementation, businesses can minimize risks and allocate resources more efficiently.

2. Industries Using A/B Testing:

E-commerce: Companies like Amazon and eBay use A/B testing to optimize website layouts, product descriptions, and checkout processes to improve sales and user experience.

Tech: Tech companies such as Google, Facebook, and Netflix utilize A/B testing extensively to enhance user engagement, feature adoption, and ad effectiveness.

Digital Marketing: A/B testing is a fundamental tool for digital marketers to optimize email marketing campaigns, ad creatives, landing pages, and calls-to-action (CTAs).

Finance: Financial institutions employ A/B testing to refine user interfaces, customer onboarding processes, and investment strategies to enhance user satisfaction and retention.

1.2 Use Cases of A/B Testing Across Various Industries:

1. Website Optimization:

Healthcare: Testing different layouts and content formats on healthcare websites to improve patient engagement and appointment scheduling.

Education: Experimenting with website designs and course descriptions to increase enrollment rates and student engagement.

Banking: Testing user interface elements and navigation paths on banking websites to enhance online banking experiences and promote financial literacy.

Energy: Experimenting with website messaging and call-to-action buttons on energy provider websites to encourage energy-saving behaviors among customers.

Government: Testing different website designs and information architectures on government portals to streamline access to public services and information.

2. Email Marketing:

Healthcare: Testing email subject lines and content variations in healthcare newsletters to increase patient education and promote healthy behaviors.

Education: Experimenting with email frequency and personalized recommendations in educational newsletters to enhance student retention and academic success.

Banking: Testing email campaign timing and messaging to promote new financial products and services among bank customers.

Energy: Experimenting with energy-saving tips and incentives in email communications to encourage energy conservation among consumers.

Government: Testing email outreach strategies to increase citizen engagement in government initiatives and public participation.

3. Mobile App Development:

Healthcare: Testing different features and user interface designs in health monitoring apps to improve user engagement and adherence to treatment plans.

Education: Experimenting with gamification elements and learning modules in educational apps to enhance student motivation and learning outcomes.

Banking: Testing app navigation flows and security features in mobile banking apps to ensure seamless and secure access to financial services.

Energy: Experimenting with energy tracking tools and personalized recommendations in energy management apps to empower consumers to make informed energy-saving decisions.

Government: Testing mobile app functionalities and user interfaces in government service apps to facilitate convenient access to public services and information.

4. Ad Campaign Optimization:

Healthcare: Testing ad creatives and targeting parameters in healthcare advertising campaigns to promote preventive care services and health screenings.

Education: Experimenting with ad messaging and audience segmentation in educational advertising campaigns to attract prospective students and donors.

Banking: Testing ad placements and messaging on digital platforms to increase brand awareness and drive customer acquisition for banking products.

Energy: Experimenting with ad formats and messaging in energy conservation campaigns to raise awareness about renewable energy solutions and energy efficiency programs.

Government: Testing ad content and targeting strategies in government public awareness campaigns to promote civic engagement and community involvement.

A/B testing *is a versatile and valuable tool that can be applied across diverse industries and sectors to drive optimization, innovation, and better outcomes for stakeholders and society as a whole.*

1.3 Dataset Overview: A/B Testing Case Study

A/B testing, also known as split testing, is a powerful technique used by businesses to compare two or more versions of a product or marketing campaign to determine which one performs better. In this case study, we analyze a dataset sourced from **Kaggle**, submitted by **İlker Yıldız**, which focuses on A/B testing in the context of online advertising.

Dataset Features:

Campaign Name: The name of the advertising campaign.

Date: The date of the record.

Spend: The amount spent on the campaign in dollars.

Number of Impressions: The number of impressions the ad received during the campaign.

Reach: The number of unique impressions received by the ad.

Website Clicks: The number of clicks received by the website through the ads.

Searches Received: The number of users who performed searches on the website.

Content Viewed: The number of users who viewed content and products on the website.

Added to Cart: The number of users who added products to their cart.

Purchases: The number of purchases made as a result of the campaign.

Campaign Types:

The dataset contains information about two types of campaigns:

Control Campaign: Represents the existing bidding type, referred to as "maximum bidding".

Test Campaign: Introduces a new bidding type, known as "average bidding", as an alternative.

Objective:

The company aims to understand if the new bidding type, **"average bidding"**, leads to more conversions compared to the existing **"maximum bidding"** strategy. The **A/B test** ran for one month, and the results need to be analyzed and presented to the client for decision-making.

Dataset Source:

The dataset was sourced from **Kaggle** and consists of records from the A/B test conducted by the company.

For further exploration and analysis, you can access the dataset from here

Loading Relevant Libraries:

[1]:
```python
# Importing the pandas library as pd for data manipulation and analysis.
import pandas as pd

# Importing Python's built-in datetime module to work with dates and times.
import datetime

# Importing specific classes from the datetime module for handling dates and␣
↪performing date arithmetic.
from datetime import date, timedelta

# Importing Plotly's Graph Objects module as go for creating customizable plots␣
↪with detailed control.
```

```
import plotly.graph_objects as go

# Importing Plotly Express as px for creating a wide array of visualizations␣
↪with minimal code.
import plotly.express as px

# Importing Plotly's Input/Output module as pio for configuring output settings␣
↪and rendering plots.
import plotly.io as pio

# Setting the default plotting template to "plotly_white" for a clean and␣
↪distraction-free background in all plots.
pio.templates.default = "plotly_white"
```

Pandas Import: The pandas library is essential for data manipulation tasks such as reading, filtering, and transforming data. It's commonly used to work with data in DataFrame formats, which are tabular data structures similar to Excel spreadsheets.

Datetime Import: The datetime module is included to handle all functionalities related to date and time, such as manipulating dates or calculating differences between dates.

Datetime Classes Import: Direct imports of date and timedelta are used for creating date objects and for conducting operations that involve durations (e.g., adding days to a date).

Plotly Graph Objects: This import allows for precise control over the components of a plot, useful for when you need to make specific adjustments that are not covered by higher-level tools.

Plotly Express: A tool for quickly and efficiently creating a broad range of plots with less code and fewer adjustments needed for a visually pleasing result.

Plotly IO: This module's functionality is geared towards managing how plots are displayed and saved, which is crucial for producing the output in the desired format or style.

Template Setting: Setting the default template to "plotly_white" ensures that all plots created in this session have a consistent, clean aesthetic, enhancing readability and visual appeal.

Data Loading:

[2]:
```
control_data = pd.read_csv("control_group.csv", sep = ";")
test_data = pd.read_csv("test_group.csv", sep = ";")
```

Import Statement: The import pandas as pd line is crucial as it brings the pandas library into the script, enabling the use of its powerful data manipulation functions. Pandas is widely used in data science for handling and analyzing data in DataFrame and Series formats.

Reading CSV Files: The pd.read_csv() function is used twice here to load data from two separate CSV files (control_group.csv and test_group.csv). This function transforms the data in these files into DataFrame objects, which are versatile data structures similar to tables in relational databases or Excel spreadsheets.

File Names and Paths: Each call to pd.read_csv() specifies the name of the CSV file to be read. These files are expected to be in the same directory as the script unless a path is otherwise specified.

Delimiter Specification (sep=";"): Both CSV files use a semicolon as a delimiter, a critical piece of information for correctly parsing the files. The delimiter separates the columns in each row of the file. Specifying the correct delimiter ensures that the data is split into columns as intended.

Let's have a look at both datasets:

Data Exploration:

[3]:
```
control_data.head()
```

[3]:
```
      Campaign Name       Date  Spend [USD]  # of Impressions     Reach  \
0  Control Campaign  1.08.2019         2280           82702.0   56930.0
1  Control Campaign  2.08.2019         1757          121040.0  102513.0
2  Control Campaign  3.08.2019         2343          131711.0  110862.0
3  Control Campaign  4.08.2019         1940           72878.0   61235.0
4  Control Campaign  5.08.2019         1835               NaN       NaN

   # of Website Clicks  # of Searches  # of View Content  # of Add to Cart  \
0               7016.0         2290.0             2159.0            1819.0
1               8110.0         2033.0             1841.0            1219.0
2               6508.0         1737.0             1549.0            1134.0
3               3065.0         1042.0              982.0            1183.0
4                  NaN            NaN                NaN               NaN

   # of Purchase
0          618.0
1          511.0
2          372.0
3          340.0
4            NaN
```

The **control_data.head()** function in Python is used to display the first five rows of the DataFrame control_data.

This method is useful for quickly examining the data's structure, including column names and initial values, which helps in understanding the dataset at a glance. It's particularly valuable in the early stages of data analysis for identifying data issues or confirming data integrity.

[4]:
```
control_data.describe()
```

[4]:
```
       Spend [USD]  # of Impressions          Reach  # of Website Clicks  \
count    30.000000         29.000000      29.000000            29.000000
mean   2288.433333     109559.758621   88844.931034          5320.793103
std     367.334451      21688.922908   21832.349595          1757.369003
min    1757.000000      71274.000000   42859.000000          2277.000000
25%    1945.500000      92029.000000   74192.000000          4085.000000
50%    2299.500000     113430.000000   91579.000000          5224.000000
75%    2532.000000     121332.000000  102479.000000          6628.000000
```

```
max    3083.000000     145248.000000  127852.000000           8137.000000

       # of Searches  # of View Content  # of Add to Cart  # of Purchase
count      29.000000          29.000000         29.000000      29.000000
mean     2221.310345        1943.793103       1300.000000     522.793103
std       866.089368         777.545469        407.457973     185.028642
min      1001.000000         848.000000        442.000000     222.000000
25%      1615.000000        1249.000000        930.000000     372.000000
50%      2390.000000        1984.000000       1339.000000     501.000000
75%      2711.000000        2421.000000       1641.000000     670.000000
max      4891.000000        4219.000000       1913.000000     800.000000
```

The **control_data.describe()** function in Python is used to display summary statistics of a DataFrame called control_data. It provides key metrics such as mean, standard deviation, minimum, and maximum values, as well as the quartiles for numerical columns, which helps in quickly understanding the distribution and central tendencies of the data.

[5]:
```
test_data.head()
```

[5]:
```
   Campaign Name       Date  Spend [USD]  # of Impressions  Reach  \
0  Test Campaign  1.08.2019         3008             39550  35820
1  Test Campaign  2.08.2019         2542            100719  91236
2  Test Campaign  3.08.2019         2365             70263  45198
3  Test Campaign  4.08.2019         2710             78451  25937
4  Test Campaign  5.08.2019         2297            114295  95138

   # of Website Clicks  # of Searches  # of View Content  # of Add to Cart  \
0                 3038           1946               1069               894
1                 4657           2359               1548               879
2                 7885           2572               2367              1268
3                 4216           2216               1437               566
4                 5863           2106                858               956

   # of Purchase
0            255
1            677
2            578
3            340
4            768
```

[6]:
```
test_data.describe()
```

[6]:
```
       Spend [USD]  # of Impressions         Reach  # of Website Clicks  \
count    30.000000         30.000000     30.000000            30.000000
mean   2563.066667      74584.800000  53491.566667          6032.333333
std     348.687681      32121.377422  28795.775752          1708.567263
min    1968.000000      22521.000000  10598.000000          3038.000000
25%    2324.500000      47541.250000  31516.250000          4407.000000
```

```
50%     2584.000000       68853.500000   44219.500000          6242.500000
75%     2836.250000       99500.000000   78778.750000          7604.750000
max     3112.000000      133771.000000  109834.000000          8264.000000

       # of Searches  # of View Content  # of Add to Cart  # of Purchase
count      30.000000          30.000000         30.000000      30.000000
mean     2418.966667        1858.000000        881.533333     521.233333
std       388.742312         597.654669        347.584248     211.047745
min      1854.000000         858.000000        278.000000     238.000000
25%      2043.000000        1320.000000        582.500000     298.000000
50%      2395.500000        1881.000000        974.000000     500.000000
75%      2801.250000        2412.000000       1148.500000     701.000000
max      2978.000000        2801.000000       1391.000000     890.000000
```

1.3.1 Data Preparation

The datasets have some errors in column names. Let's give new column names before moving forward:

[7]:
```python
# Renaming the columns of the DataFrame 'control_data' to have meaningful names.
# This helps in making the data easier to reference and work with in analysis␣
 ↪scripts.
control_data.columns = ["Campaign Name", "Date", "Amount Spent",
                        "Number of Impressions", "Reach", "Website Clicks",
                        "Searches Received", "Content Viewed", "Added to Cart",
                        "Purchases"]

# Similarly, renaming the columns of the DataFrame 'test_data' to match those␣
 ↪of 'control_data'.
# Consistent column names across DataFrames simplify data manipulation and␣
 ↪comparison.
test_data.columns = ["Campaign Name", "Date", "Amount Spent",
                     "Number of Impressions", "Reach", "Website Clicks",
                     "Searches Received", "Content Viewed", "Added to Cart",
                     "Purchases"]
```

Column Naming: The code assigns a new list of column names to the columns attribute of both control_data and test_data DataFrames. This is done by directly setting the columns property to a list of string names that describe the data in each column more clearly.

Purpose:

Clarity and Ease of Use: By renaming the columns, the data becomes easier to work with, reference, and understand. For instance, names like **"Amount Spent"** or **"Website Clicks"** clearly tell what the data in these columns represent.

Consistency Across DataFrames: Ensuring both DataFrames have the same column names allows for easier data manipulation, such as merging, comparing, and analyzing data across both the control and test groups.

Columns Defined:

“Campaign Name”: Name or identifier of the marketing campaign.

“Date”: The date on which the data was recorded.

“Amount Spent”: Financial expenditure on the campaign.

“Number of Impressions”: The count of how often the campaign’s ads were viewed.

“Reach”: The number of unique users who saw the ads.

“Website Clicks”: The number of clicks on the campaign’s ads leading to the website.

“Searches Received”: The number of searches generated from the campaign.

“Content Viewed”: The amount of campaign content viewed by the audience.

“Added to Cart”: The number of times items were added to shopping carts from the campaign.

“Purchases”: The number of purchases made resulting from the campaign.

Now let’s see if the datasets have null values or not:

[8]:
```
print(control_data.isnull().sum())
```

```
Campaign Name            0
Date                     0
Amount Spent             0
Number of Impressions    1
Reach                    1
Website Clicks           1
Searches Received        1
Content Viewed           1
Added to Cart            1
Purchases                1
dtype: int64
```

control_data.isnull().sum() is used to count the number of missing (null) values in each column of the DataFrame control_data. Here’s a brief explanation:

control_data: This is the DataFrame containing your data.

.isnull(): This method returns a DataFrame of the same shape as control_data with boolean values indicating where missing values (NaN or None) are present. True indicates a missing value, while False indicates a non-missing value.

.sum(): This function is then applied to the DataFrame returned by .isnull() to count the number of True values (missing values) in each column.

Output Interpretation:

When you execute **print(control_data.isnull().sum()),** you will get a series where the index represents the column names of **control_data,** and the values represent the count of missing values in each column. This information is useful for identifying columns with missing data, which may require cleaning or imputation before further analysis.

[9]:
```
print(test_data.isnull().sum())
```

```
Campaign Name          0
Date                   0
Amount Spent           0
Number of Impressions  0
Reach                  0
Website Clicks         0
Searches Received      0
Content Viewed         0
Added to Cart          0
Purchases              0
dtype: int64
```

The dataset of the control campaign has missing values in a row. Let's fill in these missing values by the mean value of each column:

[10]:
```
control_data["Number of Impressions"].fillna(value=control_data["Number of␣
 ↪Impressions"].mean(),
                                             inplace=True)
control_data["Reach"].fillna(value=control_data["Reach"].mean(),
                             inplace=True)
control_data["Website Clicks"].fillna(value=control_data["Website Clicks"].
 ↪mean(),
                                      inplace=True)
control_data["Searches Received"].fillna(value=control_data["Searches␣
 ↪Received"].mean(),
                                         inplace=True)
control_data["Content Viewed"].fillna(value=control_data["Content Viewed"].
 ↪mean(),
                                      inplace=True)
control_data["Added to Cart"].fillna(value=control_data["Added to Cart"].mean(),
                                     inplace=True)
control_data["Purchases"].fillna(value=control_data["Purchases"].mean(),
                                 inplace=True)
```

For each specified column in **control_data,** missing values **(NaN)** are filled using the **fillna()** method.

The value parameter is set to the mean value of each respective column. This is achieved by accessing the mean value of each column using **.mean().**

The **inplace=True** parameter ensures that the changes are applied directly to the **control_data** DataFrame without the need for reassignment.

Purpose:

Data Imputation: Filling missing values with the mean is a common technique in data preprocessing to handle missing data. It helps in preserving the overall distribution of the data while mitigating the impact of missing values on subsequent analyses.

Maintaining Data Integrity: By replacing missing values with the mean, the integrity and completeness of the dataset are maintained, allowing for more robust analysis and modeling. However, it's important to note that imputing missing values with the mean may introduce bias, especially if the missingness is related to specific patterns in the data. Other imputation techniques like median imputation or using predictive models may be more appropriate depending on the specific context of your company or organization research purpose.

Now I will create a new dataset by merging both datasets:

```
[11]: # Specify the date format explicitly for conversion
      date_format = "%d.%m.%Y"  # This corresponds to 'day.month.year'
```

```
[12]: # Convert the 'Date' columns in both DataFrames to datetime objects using the␣
       ↪specified format
      control_data['Date'] = pd.to_datetime(control_data['Date'], format=date_format)
      test_data['Date'] = pd.to_datetime(test_data['Date'], format=date_format)
```

```
[13]: # Perform the merge on the 'Date' column, sort by 'Date', and reset the index
      #ab_data = control_data.merge(test_data, on='Date', how='outer').
       ↪sort_values('Date')
      #ab_data = ab_data.reset_index(drop=True)

      # Concatenate the DataFrames to stack them vertically
      ab_data = pd.concat([control_data, test_data], ignore_index=True)

      # If you need to sort by 'Date' post-concatenation
      ab_data = ab_data.sort_values(by='Date').reset_index(drop=True)
```

```
[14]: # Display the first few rows and the structure of the DataFrame
      ab_data.head()
```

```
[14]:       Campaign Name       Date  Amount Spent  Number of Impressions     Reach  \
      0  Control Campaign 2019-08-01          2280                82702.0   56930.0
      1     Test Campaign 2019-08-01          3008                39550.0   35820.0
      2  Control Campaign 2019-08-02          1757               121040.0  102513.0
      3     Test Campaign 2019-08-02          2542               100719.0   91236.0
      4  Control Campaign 2019-08-03          2343               131711.0  110862.0

         Website Clicks  Searches Received  Content Viewed  Added to Cart  Purchases
      0          7016.0             2290.0          2159.0         1819.0      618.0
      1          3038.0             1946.0          1069.0          894.0      255.0
      2          8110.0             2033.0          1841.0         1219.0      511.0
      3          4657.0             2359.0          1548.0          879.0      677.0
      4          6508.0             1737.0          1549.0         1134.0      372.0
```

```
[15]: ab_data.describe()
```

[15]:
```
                        Date  Amount Spent  Number of Impressions  \
count                     60     60.000000              60.000000
mean     2019-08-15 12:00:00   2425.750000           92072.279310
min      2019-08-01 00:00:00   1757.000000           22521.000000
25%      2019-08-08 00:00:00   2073.750000           69558.250000
50%      2019-08-15 12:00:00   2420.500000           98281.000000
75%      2019-08-23 00:00:00   2727.500000          117160.500000
max      2019-08-30 00:00:00   3112.000000          145248.000000
std                      NaN    381.130461           32270.541283

               Reach  Website Clicks  Searches Received  Content Viewed  \
count      60.000000       60.000000          60.000000       60.000000
mean    71168.248851     5676.563218        2320.138506     1900.896552
min     10598.000000     2277.000000        1001.000000      848.000000
25%     43235.500000     4230.750000        1970.750000     1249.000000
50%     77422.000000     5581.000000        2374.500000     1959.396552
75%     95314.250000     7201.250000        2755.750000     2422.500000
max    127852.000000     8264.000000        4891.000000     4219.000000
std     30847.039691     1740.469866         663.473391      681.437956

       Added to Cart   Purchases
count      60.000000   60.000000
mean     1090.766667  522.013218
min       278.000000  222.000000
25%       863.250000  340.000000
50%      1082.500000  506.000000
75%      1384.250000  685.000000
max      1913.000000  890.000000
std       427.427479  195.297540
```

Before moving forward, let's have a look if the dataset has an equal number of samples about both campaigns:

[16]:
```python
ab_data = pd.concat([control_data, test_data], ignore_index=True)

# If you need to sort by 'Date' post-concatenation
ab_data = ab_data.sort_values(by='Date').reset_index(drop=True)

# Display the first few rows and the structure of the DataFrame
print(ab_data.head())
print("Columns in concatenated DataFrame:", ab_data.columns)
```

```
      Campaign Name       Date  Amount Spent  Number of Impressions     Reach  \
0  Control Campaign 2019-08-01          2280                82702.0   56930.0
1     Test Campaign 2019-08-01          3008                39550.0   35820.0
2  Control Campaign 2019-08-02          1757               121040.0  102513.0
3     Test Campaign 2019-08-02          2542               100719.0   91236.0
4  Control Campaign 2019-08-03          2343               131711.0  110862.0
```

```
   Website Clicks  Searches Received  Content Viewed  Added to Cart  Purchases
0          7016.0             2290.0          2159.0         1819.0      618.0
1          3038.0             1946.0          1069.0          894.0      255.0
2          8110.0             2033.0          1841.0         1219.0      511.0
3          4657.0             2359.0          1548.0          879.0      677.0
4          6508.0             1737.0          1549.0         1134.0      372.0
Columns in concatenated DataFrame: Index(['Campaign Name', 'Date', 'Amount
Spent', 'Number of Impressions',
       'Reach', 'Website Clicks', 'Searches Received', 'Content Viewed',
       'Added to Cart', 'Purchases'],
      dtype='object')
```

[17]:
```python
# Print the columns to see what's included after merging
print("Columns in merged DataFrame:", ab_data.columns)
```

```
Columns in merged DataFrame: Index(['Campaign Name', 'Date', 'Amount Spent',
'Number of Impressions',
       'Reach', 'Website Clicks', 'Searches Received', 'Content Viewed',
       'Added to Cart', 'Purchases'],
      dtype='object')
```

[18]:
```python
print(ab_data["Campaign Name"].value_counts())
```

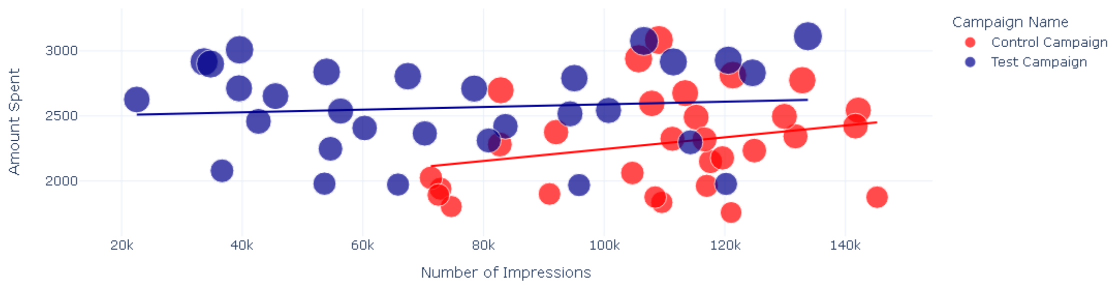
```
Campaign Name
Control Campaign    30
Test Campaign       30
Name: count, dtype: int64
```

The dataset has 30 samples for each campaign. Now let's start with A/B testing to find the best marketing strategy.

A/B Testing to Find the Best Marketing Strategy

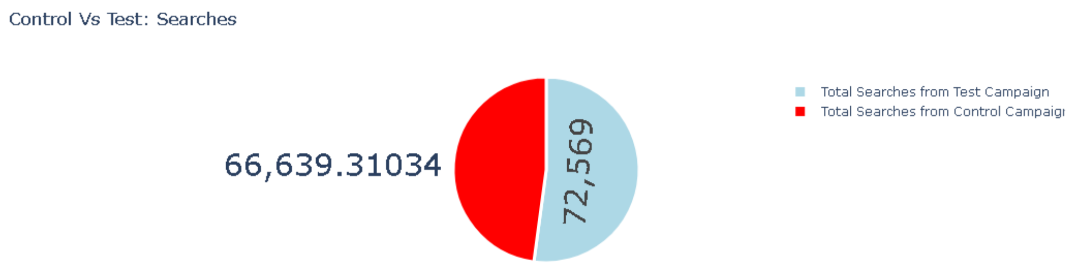
To kick off A/B testing, I'll start by examining how the number of impressions relates to the amount spent on each campaign. This analysis will provide insights into campaign effectiveness and inform future testing strategies.

[19]:
```python
# Create a scatter plot and specify colors directly for the known campaigns
figure = px.scatter(
    data_frame=ab_data,
    x="Number of Impressions",
    y="Amount Spent",
    size="Amount Spent",  # Adjust point size based on 'Amount Spent'
    color="Campaign Name",  # Color points by 'Campaign Name'
    color_discrete_map={  # Directly specify colors for specific campaigns
        'Control Campaign': 'red',  # Light blue for the Control Campaign
        'Test Campaign': 'darkblue',      # Dark blue for the Test Campaign
        # Add more campaigns with their colors if there are more than two
    },
    trendline="ols"  # Add an ordinary least squares regression line
)

# Show the plot
figure.show()
```

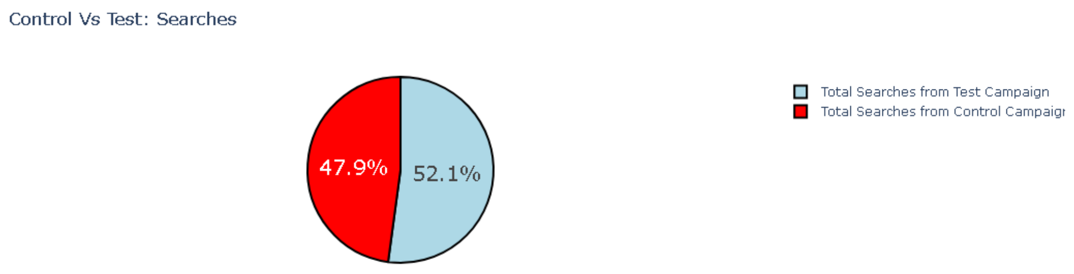


The control campaign generated higher impressions compared to the amount spent on both campaigns. Next, let's examine the number of searches conducted on the website from both campaigns

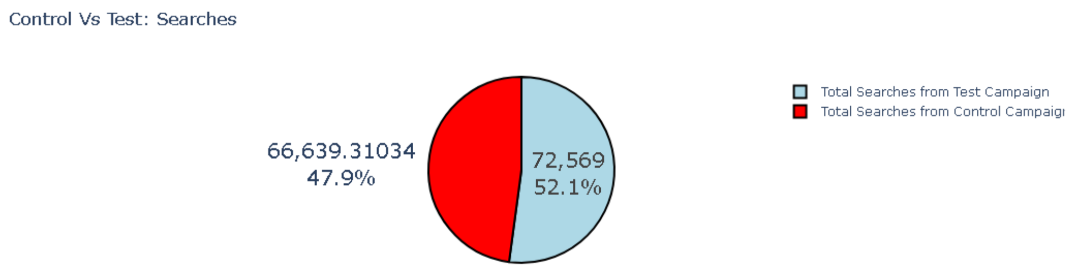
[20]:
```
# Labels for the pie chart
label = ["Total Searches from Control Campaign",
         "Total Searches from Test Campaign"]

# Calculating the total searches received for each campaign
counts = [sum(control_data["Searches Received"]),
          sum(test_data["Searches Received"])]

# Colors for each segment of the pie chart
colors = ['red', 'lightblue']  # Using red for Control and lightblue for Test

# Create a pie chart
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])

# Update the layout and trace properties
fig.update_layout(title_text='Control Vs Test: Searches')
fig.update_traces(
    hoverinfo='label+percent',
    textinfo='value',
    textfont_size=30,
    marker=dict(colors=colors, line=dict(color='white', width=3))
)

# Display the pie chart
fig.show()
```



[21]:
```python
# Labels for the pie chart
label = ["Total Searches from Control Campaign",
         "Total Searches from Test Campaign"]

# Calculating the total searches received for each campaign
counts = [sum(control_data["Searches Received"]),
          sum(test_data["Searches Received"])]

# Colors for each segment of the pie chart
colors = ['red', 'lightblue']  # Using red for Control and lightblue for Test

# Create a pie chart
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])

# Update the layout and trace properties to show percentage
fig.update_layout(title_text='Control Vs Test: Searches')
fig.update_traces(
    hoverinfo='label+percent',
    textinfo='percent',  # Updated to show percentage
    textfont_size=20,
    marker=dict(colors=colors, line=dict(color='black', width=2))
)

# Display the pie chart
fig.show()
```

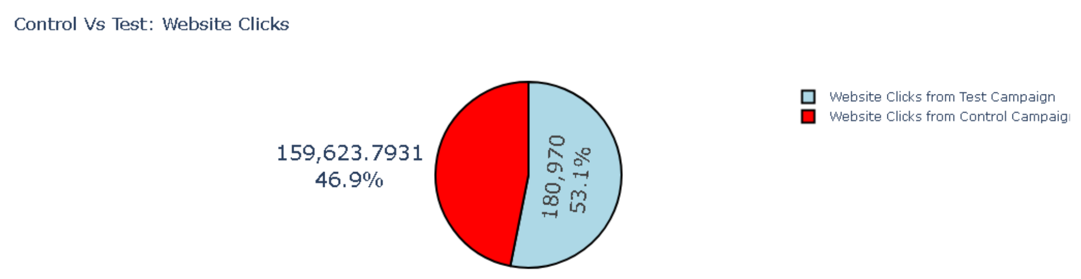


[22]:
```python
# Labels for the pie chart
label = ["Total Searches from Control Campaign",
         "Total Searches from Test Campaign"]

# Calculating the total searches received for each campaign
counts = [sum(control_data["Searches Received"]),
          sum(test_data["Searches Received"])]

# Colors for each segment of the pie chart
colors = ['red', 'lightblue']  # Using red for Control and lightblue for Test

# Create a pie chart
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])

# Update the layout and trace properties to show both number and percentage
fig.update_layout(title_text='Control Vs Test: Searches')
fig.update_traces(
    hoverinfo='label+percent',
    textinfo='value+percent',  # Show both the value and the percentage
    textfont_size=20,
    marker=dict(colors=colors, line=dict(color='black', width=2))
)

# Display the pie chart
fig.show()
```



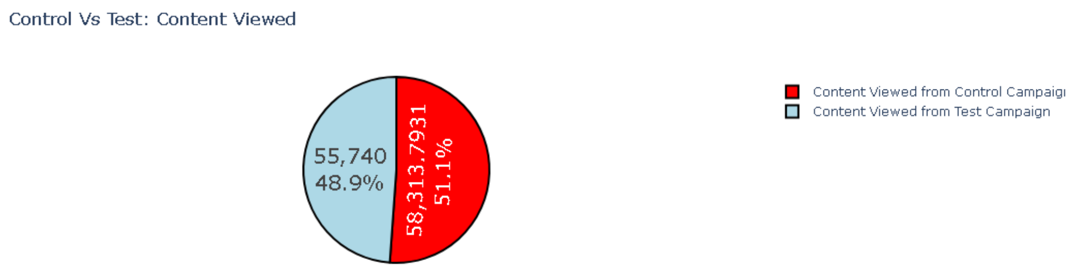
The test campaign yielded more website searches. Now, let's delve into the number of website clicks from both campaigns.

[23]:
```python
# Labels for the pie chart
label = ["Website Clicks from Control Campaign",
         "Website Clicks from Test Campaign"]

# Calculating the total website clicks received for each campaign
counts = [sum(control_data["Website Clicks"]),
          sum(test_data["Website Clicks"])]

# Colors for each segment of the pie chart
colors = ['red', 'lightblue']  # Using red for Control and lightblue for Test

# Create a pie chart
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])

# Update the layout and trace properties to show both number and percentage
fig.update_layout(title_text='Control Vs Test: Website Clicks')
fig.update_traces(
    hoverinfo='label+percent',
    textinfo='value+percent',  # Now showing both the value and the percentage
    textfont_size=20,
    marker=dict(colors=colors, line=dict(color='black', width=2))
)

# Display the pie chart
fig.show()
```



The test campaign leads in the number of website clicks. Now, let's examine the amount of content viewed after visitors reached the website from both campaigns.

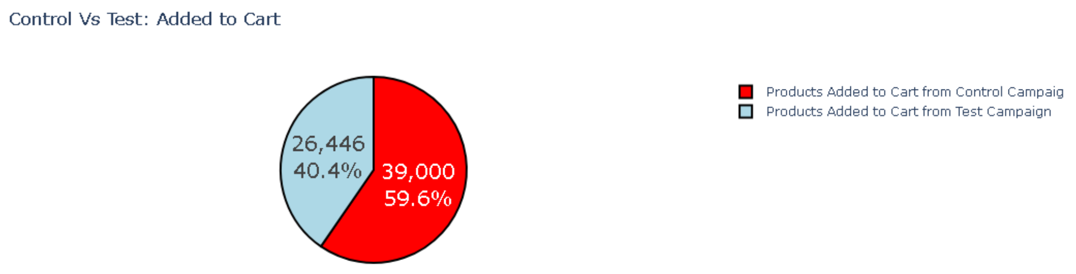
[24]:
```
# Labels for the pie chart
label = ["Content Viewed from Control Campaign",
         "Content Viewed from Test Campaign"]

# Calculating the total content viewed received for each campaign
counts = [sum(control_data["Content Viewed"]),
          sum(test_data["Content Viewed"])]

# Colors for each segment of the pie chart
colors = ['red', 'lightblue']  # Using red for Control and lightblue for Test

# Create a pie chart
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])

# Update the layout and trace properties to show both number and percentage
fig.update_layout(title_text='Control Vs Test: Content Viewed')
fig.update_traces(
    hoverinfo='label+percent',
    textinfo='value+percent',  # Show both the value and the percentage
    textfont_size=20,
    marker=dict(colors=colors, line=dict(color='black', width=2))
)
```



The control campaign's audience viewed more content compared to the test campaign. Despite the slight variance, considering the lower website clicks of the control campaign, its engagement on the website exceeds that of the test campaign.

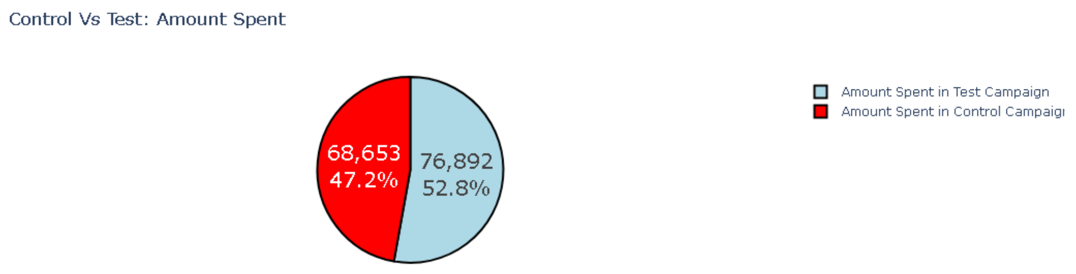
Now, let's shift our focus to the number of products added to the cart from both campaigns.

[25]:
```
# Labels for the pie chart
label = ["Products Added to Cart from Control Campaign",
         "Products Added to Cart from Test Campaign"]

# Calculating the total products added to cart for each campaign
counts = [sum(control_data["Added to Cart"]),
          sum(test_data["Added to Cart"])]

# Colors for each segment of the pie chart, updating to new color preferences
# Colors for each segment of the pie chart
colors = ['red', 'lightblue']  # Using red for Control and lightblue for Test


# Create a pie chart
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])

# Update the layout and trace properties to show both number and percentage
fig.update_layout(title_text='Control Vs Test: Added to Cart')
fig.update_traces(
    hoverinfo='label+percent',
    textinfo='value+percent',  # Now showing both the value and the percentage
    textfont_size=20,  # Adjusted text size for better visibility
    marker=dict(colors=colors, line=dict(color='black', width=2))  # Adjusted␣
 ↪line width
)

# Display the pie chart
fig.show()
```



Despite the low website clicks, more products were added to the cart from the control campaign. Now, let's examine the amount spent on both campaigns.

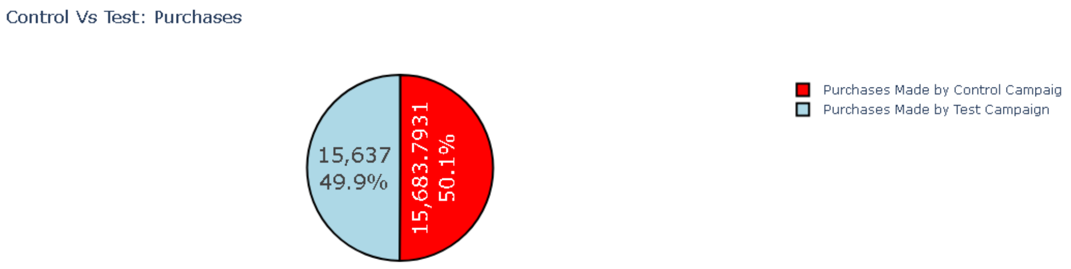
[26]:
```python
# Labels for the pie chart
label = ["Amount Spent in Control Campaign",
         "Amount Spent in Test Campaign"]

# Calculating the total amount spent for each campaign
counts = [sum(control_data["Amount Spent"]),
          sum(test_data["Amount Spent"])]

# Colors for each segment of the pie chart, using blue for Control and red for␣
 ↪Test
colors = ['red', 'lightblue']  # Using red for Control and lightblue for Test

# Create a pie chart
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])

# Update the layout and trace properties to show both number and percentage
fig.update_layout(title_text='Control Vs Test: Amount Spent')
fig.update_traces(
    hoverinfo='label+percent',
    textinfo='value+percent',  # Showing both the value and the percentage
    textfont_size=20,  # Adjusted text size for better readability
    marker=dict(colors=colors, line=dict(color='black', width=2))  # Adjusted␣
 ↪line width
)

# Display the pie chart
fig.show()
```



While the amount spent on the test campaign exceeds that of the control campaign, it's noteworthy that the control campaign generated more content views and product additions to the cart. Hence, despite the higher expenditure, the control campaign appears to be more efficient than the test campaign.

Now, let's examine the purchases made by both campaigns.

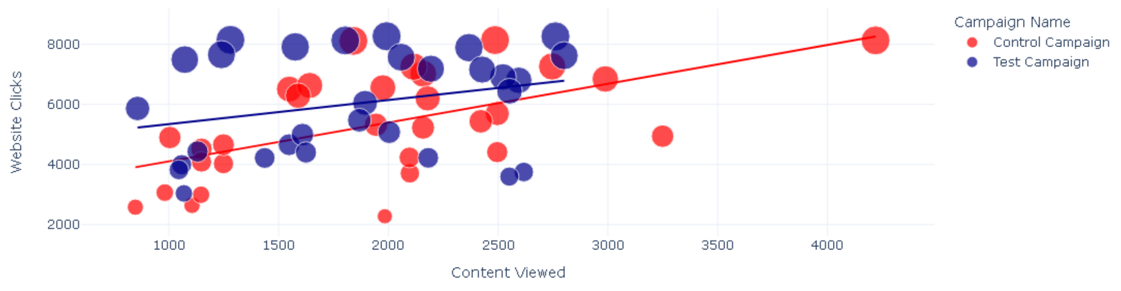
[27]:
```
# Labels for the pie chart
label = ["Purchases Made by Control Campaign",
         "Purchases Made by Test Campaign"]

# Calculating the total purchases made by each campaign
counts = [sum(control_data["Purchases"]),
          sum(test_data["Purchases"])]

# Colors for each segment of the pie chart, switching to blue for Control and␣
↪red for Test
colors = ['red', 'lightblue']  # Using red for Control and lightblue for Test

# Create a pie chart
fig = go.Figure(data=[go.Pie(labels=label, values=counts)])

# Update the layout and trace properties to show both number and percentage
fig.update_layout(title_text='Control Vs Test: Purchases')
fig.update_traces(
    hoverinfo='label+percent',
    textinfo='value+percent',  # Now showing both the value and the percentage
    textfont_size=20,  # Reduced font size for clarity
    marker=dict(colors=colors, line=dict(color='black', width=2))  # Reduced␣
↪line width
)

# Display the pie chart
fig.show()
```



With only a marginal difference of around 1% in the purchases made from both ad campaigns, the control campaign emerges victorious. Given that the control campaign achieved more sales with a lower marketing expenditure, it clearly outperforms the test campaign in this regard!

Now, let's analyze some metrics to determine which ad campaign converts more effectively. To begin, I'll explore the relationship between the number of website clicks and content viewed from both campaigns.

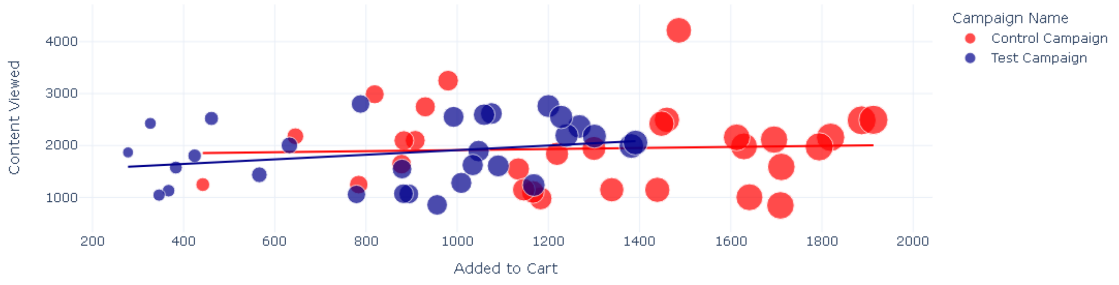
[28]:
```python
# Create a scatter plot analyzing the relationship between content viewed and␣
 ↪website clicks
figure = px.scatter(
    data_frame=ab_data,  # Specify the DataFrame containing the data
    x="Content Viewed",  # Set 'Content Viewed' as the x-axis
    y="Website Clicks",  # Set 'Website Clicks' as the y-axis
    size="Website Clicks",  # Use 'Website Clicks' to determine the size of the␣
 ↪scatter points
    color="Campaign Name",  # Color code the points by 'Campaign Name' for␣
 ↪differentiation
    color_discrete_map={  # Directly specify colors for specific campaigns
        'Control Campaign': 'red',  # Use red for the Control Campaign
        'Test Campaign': 'darkblue'  # Use dark blue for the Test Campaign
        # Add more campaigns and their colors if necessary
    },
    trendline="ols"  # Add an ordinary least squares regression line to␣
 ↪indicate trends
)

# Display the plot
figure.show()
```



While the website clicks are higher in the test campaign, the engagement stemming from these clicks is higher in the control campaign. Thus, the control campaign emerges as the winner in terms of engagement!

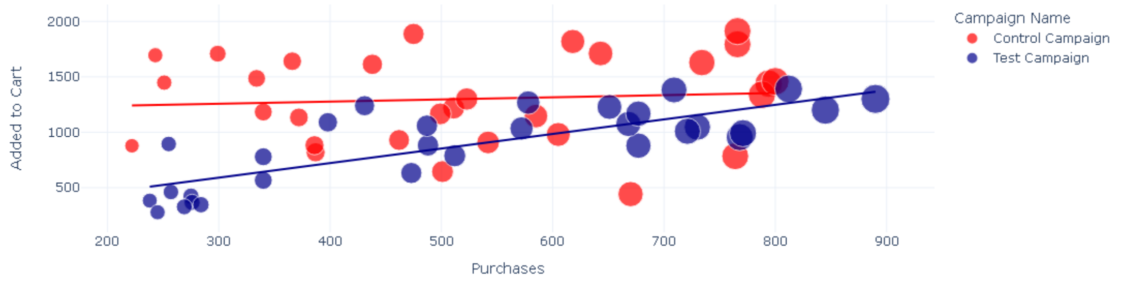
Now, I'll analyze the relationship between the amount of content viewed and the number of products added to the cart from both campaigns.

[29]:
```
# Create a scatter plot analyzing the relationship between 'Added to Cart' and␣
↪'Content Viewed'
figure = px.scatter(
    data_frame=ab_data,  # Specify the DataFrame containing the data
    x="Added to Cart",  # Set 'Added to Cart' as the x-axis
    y="Content Viewed",  # Set 'Content Viewed' as the y-axis
    size="Added to Cart",  # Use 'Added to Cart' to determine the size of the␣
↪scatter points
    color="Campaign Name",  # Color code the points by 'Campaign Name' for␣
↪differentiation
    color_discrete_map={  # Directly specify colors for specific campaigns
        'Control Campaign': 'red',  # Use red for the Control Campaign
        'Test Campaign': 'darkblue'  # Use dark blue for the Test Campaign
        # Add more campaigns and their colors if necessary
    },
    trendline="ols"  # Add an ordinary least squares regression line to␣
↪indicate trends
)

# Display the plot
figure.show()
```



Once more, the control campaign emerges as the winner! Now, let's delve into the relationship between the number of products added to the cart and the number of sales from both campaigns.

[30]:
```
# Create a scatter plot to analyze the relationship between 'Purchases' and␣
↪'Added to Cart'
figure = px.scatter(
    data_frame=ab_data,  # Specify the DataFrame containing the data
    x="Purchases",  # Set 'Purchases' as the x-axis
    y="Added to Cart",  # Set 'Added to Cart' as the y-axis
    size="Purchases",  # Use 'Purchases' to determine the size of the scatter␣
↪points
    color="Campaign Name",  # Color code the points by 'Campaign Name' for␣
↪differentiation
    color_discrete_map={  # Directly specify colors for specific campaigns
        'Control Campaign': 'red',  # Use red for the Control Campaign
        'Test Campaign': 'darkblue'  # Use dark blue for the Test Campaign
        # Add more campaigns and their colors if necessary
    },
    trendline="ols"  # Add an ordinary least squares regression line to␣
↪indicate trends
)

# Display the plot
figure.show()
```



Despite the control campaign yielding more sales and more products in the cart, it's noteworthy that the conversion rate of the test campaign is higher.

Conclusion

Based on the findings, the following recommendations are suggested for the company:

Utilize Campaigns Strategically: Leverage the control campaign for marketing multiple products to a wider audience, while employing the test campaign to target specific products to a more tailored audience segment.

Optimize Marketing Efforts: Focus on enhancing engagement metrics and conversion rates across both campaigns to maximize returns on marketing investments.

Continuous Monitoring and Adaptation: Regularly monitor campaign performance metrics and adjust strategies based on real-time data insights to ensure ongoing effectiveness.

Further analysis will involve **Hypothesis Testing.**

This **statistical technique** is crucial for validating the **observed differences** in **campaign performance** and providing insights into whether these differences are **statistically significant** or **occurred by chance.**

By **formulating hypotheses**, **conducting statistical tests**, and **interpreting the results**, we can confidently determine the effectiveness of each campaign and make informed decisions about future marketing strategies.

Stay tuned for Part Two, where we delve deeper into Hypothesis Testing to gain a comprehensive understanding of the impact of the control and test campaigns!

You can access the notebook via this GitHub Repository

My name is **David Akanji** and you can follow me on linkedin via or direct your enquiry to `akanjiolubukoladavid@gmail.com`

```
[ ]:
```