Cluster Analysis Tool

Author: Sean Honour Tan (UCID: 30094560)

Import necessary packages

In [1]:
```
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import json
import requests
import datetime
import time
from IPython.display import display
import plotly.express as px
import plotly.graph_objs as go
```

Acquire Data

In [2]:
```
def get_data():
    # Prompt user for desired timeframe
    start_input = input('Enter the start date in the format YYYY-MM-DD: ')
    end_input = input('Enter the end date in the format YYYY-MM-DD: ')
    try:
        #convert inputs to datetime object of format YYYY-MM-DD
        start_date = datetime.datetime.strptime(start_input, '%Y-%m-%d').date()
        end_date = datetime.datetime.strptime(end_input, '%Y-%m-%d').date()
    except:
        print('Invalid date format. Please enter the date in the format YYYY-MM-DD')

    desired_types = input('Enter the desired generator types, separated by a space: ').split()

    print('Start date: ' + str(start_date))
    print('End date: ' + str(end_date))
    print('Desired generator types: ' + str(desired_types))

    # pull the asset list from the AESO API
    # initialize api url
    asset_url = 'https://api.aeso.ca/report/v1/csd/generation/assets/current'
    # initialize requests header, including valid API key
    AESO_header = {'accept': 'application/json' , 'X-API-Key': 'eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJydmM2d2IiLCJpYXQiOjE3MDYwMjMyNzB9.zWQ2w5TnM9l
    # request data from AESO API, load the JSON
    res = requests.get(asset_url, headers = AESO_header)
    data = json.loads(res.text)

    # convert the JSON data into a dataframe
    json_asset_df = pd.json_normalize(data)

    # extract the 'asset_list' column from the dataframe
    asset_df = json_asset_df.loc[0, 'return.asset_list']
    # convert the 'asset_list' column into a dataframe
    asset_df = pd.DataFrame(asset_df)

    # rename the 'asset' column to 'asset_ID'
    asset_df.rename(columns = {'asset': 'asset_ID'}, inplace = True)

    # save the asset_df to a csv file
    asset_df.to_csv('generator_type.csv')

    # read in generator type csv in order to trim down the data according to user input
    generator_list = pd.read_csv('generator_type.csv')



    #initialize variables and master dataframe
    count = 1
    t = 1
    master_df = pd.DataFrame()
    date = start_date
    #iterate through the dates
    while date <= end_date:

        # initialize api url with changeable dates
        api_url = f'https://api.aeso.ca/report/v1/meritOrder/energy?startDate={date}'

        # initialize requests header, including valid API key
        AESO_header = {'accept': 'application/json' , 'X-API-Key': 'eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJydmM2d2IiLCJpYXQiOjE3MDYwMjMyNzB9.zWQ2w5

        # request data from AESO API, load the JSON
        res = requests.get(api_url, headers = AESO_header)
        data = json.loads(res.text)

        # normalize the JSON data into flat table
        df1 = pd.json_normalize(data)


        # get list of dictionaries from df
        df1_list = df1.loc[0, 'return.data']
```

```
        # convert list of dictionaries into readable and clean dataframe ready for usage
        df2 = pd.DataFrame(df1_list)

        # concatenate dataframe into master dataframe
        master_df = pd.concat([master_df,df2], axis=0, ignore_index =True)
        # print successful request message
        print('request ' + str(count) + ' successful (date: ' + str(date) + ')')
        # increment date and count for next get request
        date = date + datetime.timedelta(days=1)
        count +=1
        # wait 1 second before next request
        time.sleep(t)

    # create empty dataframe to store daily data
    daily_master_df = pd.DataFrame()

    # iterate through the master dataframe
    for index, row in master_df.iterrows():
        # if the row contains 'energy_blocks' and it is a list
        if 'energy_blocks' in row and isinstance(row['energy_blocks'], list):
            # Create a temporary dataframe from the 'energy_blocks' list
            df_temp = pd.DataFrame(row['energy_blocks'])
            # Attach the timestamp from df2 to df_temp
            df_temp['begin_dateTime_mpt'] = row['begin_dateTime_mpt']
            # Concatenate the temporary dataframe to the daily_master_df
            daily_master_df = pd.concat([daily_master_df, df_temp], axis=0, ignore_index=True)

    # Ensure generator_list['fuel_type'] is ready for case-insensitive comparison
    generator_list['fuel_type'] = generator_list['fuel_type'].str.lower()
    desired_types = [x.lower() for x in desired_types]  # Convert desired_types to lower case for case-insensitive match

    # Filter daily_master_df based on desired generator types
    if desired_types:
        # Pre-filter generator_list to include only rows with desired fuel_types
        desired_generators = generator_list[generator_list['fuel_type'].isin(desired_types)]

        # Use isin to filter daily_master_df rows where asset_ID is in desired_generators
        filtered_daily_master_df = daily_master_df[daily_master_df['asset_ID'].isin(desired_generators['asset_ID'])]

        # Check if the filtered dataframe is not empty
        if not filtered_daily_master_df.empty:
            daily_master_df = filtered_daily_master_df
        else:
            print("No matching generator types found. Returning all data.")
    else:
        print("No desired generator types specified. Returning all data.")







    # convert columns with (Y/N) to 1/0, and remove '?' from column names
    daily_master_df['dispatched'] = daily_master_df['dispatched?'].map({'Y': 1, 'N': 0})
    daily_master_df['flexible'] = daily_master_df['flexible?'].map({'Y': 1, 'N': 0})
    daily_master_df = daily_master_df[['begin_dateTime_mpt','import_or_export' ,'asset_ID' , 'block_number' , 'block_price', 'from_MW', 'to_

    # save the daily_master_df to a csv file
    daily_master_df.to_csv('daily_master_df.csv')




get_data()
daily_master_df = pd.read_csv('daily_master_df.csv')
```

```
Enter the start date in the format YYYY-MM-DD: 2023-01-01
Enter the end date in the format YYYY-MM-DD: 2023-01-10
Enter the desired generator types, separated by a space: GAS
Start date: 2023-01-01
End date: 2023-01-10
Desired generator types: ['GAS']
request 1 successful (date: 2023-01-01)
request 2 successful (date: 2023-01-02)
request 3 successful (date: 2023-01-03)
request 4 successful (date: 2023-01-04)
request 5 successful (date: 2023-01-05)
request 6 successful (date: 2023-01-06)
request 7 successful (date: 2023-01-07)
request 8 successful (date: 2023-01-08)
request 9 successful (date: 2023-01-09)
request 10 successful (date: 2023-01-10)
```

Define Elbow, Clustering, and Metric Table Functions

In [3]:

```
def elbow_method(df, features):
    # perform the elbow method to determine the optimal number of clusters
    # create a list of inertia values for each number of clusters
    inertia = []
    # loop through the number of clusters from 1 to 10
    for i in range(1, 11):
        # create a kmeans model
        kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 300, n_init = 10, random_state = 0)
        # fit the model to the features, let number of features be to be fit be the number of elements in the features list
        kmeans.fit(df[features])
```

```
        # append the inertia value to the list
        inertia.append(kmeans.inertia_)

    fig = go.Figure()

    # Add the line plot for inertia values
    fig.add_trace(go.Scatter(x=list(range(1, 11)), y=inertia,
                        mode='lines+markers',
                        name='Inertia'))

    # Add labels and title
    fig.update_layout(title='Elbow Method',
                      width=950,
                    xaxis_title='Number of clusters',
                    yaxis_title='Inertia',
                    xaxis=dict(tickmode='linear'))

    # Show the plot
    fig.show()


# function to determine the ratio of dispatched to non-dispatched generator bids in each cluster. show the ratio in fraction form.
def dispatched_ratio(df):

    # create a new dataframe to store the cluster ratios
    dispatched_ratios = pd.DataFrame()
    # loop through the clusters
    for i in range(df['cluster'].nunique()):
        # calculate the size of the dispatched and non-dispatched bids in the cluster
        dispatched = df[(df['cluster'] == i) & (df['dispatched'] == 1)].shape[0]
        non_dispatched = df[(df['cluster'] == i) & (df['dispatched'] == 0)].shape[0]
        # calculate the ratio of dispatched to non-dispatched bids
        ratio = dispatched / non_dispatched
        # add the ratio to the dataframe
        dispatched_ratios = dispatched_ratios._append(pd.DataFrame({'cluster': [i], 'ratio': [ratio]}))
    # return the dataframe
    return display(dispatched_ratios)

# function to generate a table of the cluster means, and average block price and block size for each cluster, as well as variance of block p
def cluster_means_variances(df, features):
    # Create a new dataframe to store the cluster means and variances
    cluster_stats = pd.DataFrame()

    # Loop through the clusters
    for i in df['cluster'].unique():
        cluster_data = df[df['cluster'] == i]
        stats = {'cluster': i}

        # Calculate means and variances for each feature, if the feature is block price, weight the means and the variance by the block size
        for feature in features:
            if feature == 'block_price':
                stats[f'{feature}_mean'] = np.average(cluster_data[feature], weights=cluster_data['block_size'])
                stats[f'{feature}_variance'] = np.average((cluster_data[feature] - stats[f'{feature}_mean'])**2, weights=cluster_data['block_
            else:
                stats[f'mean_{feature}'] = cluster_data[feature].mean()
                stats[f'var_{feature}'] = cluster_data[feature].var()

        # Append the stats to the cluster_stats dataframe
        cluster_stats = cluster_stats._append(stats, ignore_index=True)

    # Ensure the 'cluster' column is of integer type
    cluster_stats['cluster'] = cluster_stats['cluster'].astype(int)

    return display(cluster_stats)

# function to generate a table of the cluster sizes
def cluster_sizes(df):
    # create a new dataframe to store the cluster sizes
    cluster_sizes = pd.DataFrame()
    # loop through the clusters
    for i in range(df['cluster'].nunique()):
        # calculate the size of the cluster
        cluster_size = df[df['cluster'] == i].shape[0]
        # add the cluster size to the dataframe
        cluster_sizes = cluster_sizes._append(pd.DataFrame({'cluster': [i], 'size': [cluster_size]}))
    # return the dataframe
    return display(cluster_sizes)

# function to generate table of centroid values
def cluster_centroids(df, features):
    # Create a new DataFrame to store the cluster centroids
    cluster_centroids = pd.DataFrame(columns=['centroid'] + features)

    # Loop through the clusters
    for i in df['cluster'].unique():
        # Calculate the centroid of the cluster
        cluster_centroid = df[df['cluster'] == i][features].mean()
        # Build a row dictionary dynamically including all features
        row = {'centroid': [i]}
        for feature, value in zip(features, cluster_centroid):
            row[feature] = value
        # Append the row to the DataFrame
        cluster_centroids = cluster_centroids._append(row, ignore_index=True)

    # Display the DataFrame
    return display(cluster_centroids)
```

```python
#function to display a table of the assets in each cluster
def cluster_assets(df):
    # Loop through the clusters
    for i in df['cluster'].unique():
        # Create a new DataFrame for the cluster
        cluster_df = df[df['cluster'] == i]
        # Display the cluster number
        print(f'Cluster {i}')
        # Display the cluster DataFrame
        display(cluster_df[['asset_ID']])
        print('\n')


# function to plot the clusters and display the cluster statistics
def kmeans_cluster(df, features, nclusters, labels, title):
    # fit the kmeans model to the features, using 3 clusters
    kmeans = KMeans(n_clusters = nclusters, init = 'k-means++', max_iter = 300, n_init = 10, random_state = 0)
    # fit the model to the features
    kmeans.fit(df[features])
    # add a new column to df3 that contains the cluster labels
    df['cluster'] = kmeans.labels_

    if len(features) == 2:
        # Create a 2D scatter plot for clusters using Plotly Express
        fig = px.scatter(df, x=features[0], y=features[1], color='cluster',
                        color_continuous_scale='jet', labels={'cluster': 'Cluster'}, hover_data=['asset_ID'])

        # Add centroids to the plot using Plotly Graph Objects
        centroids = kmeans.cluster_centers_
        fig.add_trace(go.Scatter(x=centroids[:, 0], y=centroids[:, 1],
                                mode='markers', marker=dict(color='red', size=15, opacity=0.8),
                                name='Centroids'))

        # Update the layout to add title and labels
        fig.update_layout(title=dict(text = title, y = 0.95), height = 800, xaxis_title=labels[0], yaxis_title=labels[1])
        fig.update_coloraxes(colorbar_len = 0.7)

        fig.show()
        if 'dispatched' in df.columns:
            print("RATIO OF DISPATCHED TO NON-DISPATCHED BIDS IN EACH CLUSTER")
            dispatched_ratio(df)

            print("\n ________________________________________________________________________________________ \n")

        print(f"WEIGHTED MEAN AND VARIANCE OF {features[0].upper()} AND {features[1].upper()} FOR EACH CLUSTER")
        cluster_means_variances(df, features)

        print("\n ________________________________________________________________________________________ \n")

        print("SIZE OF EACH CLUSTER:")
        cluster_sizes(df)

        print("\n ________________________________________________________________________________________ \n")

        print("CENTROID VALUES FOR EACH CLUSTER:")
        cluster_centroids(df, features)

    if len(features) == 3:
    # Generate hover text that includes asset_ID and the actual feature names
        hover_text = 'asset_ID: '+ df['asset_ID'].astype(str) + '<br>' + \
                    features[0] + ': ' + df[features[0]].astype(str) + '<br>' + \
                    features[1] + ': ' + df[features[1]].astype(str) + '<br>' + \
                    features[2] + ': ' + df[features[2]].astype(str)

        # Create a 3D scatter plot for clusters
        fig = go.Figure(data=go.Scatter3d(
            x=df[features[0]],
            y=df[features[1]],
            z=df[features[2]],
            mode='markers',
            marker=dict(
                size=5,
                color=kmeans.labels_,  # set color to the cluster labels
                colorscale='jet',   # color scale
                opacity=0.8
            ),
            text=hover_text,  # Set hover text
            hoverinfo='text'  # Display custom hover text
        ))

        # Add centroids to the plot
        centroids = kmeans.cluster_centers_
        fig.add_trace(go.Scatter3d(
            x=centroids[:, 0],
            y=centroids[:, 1],
            z=centroids[:, 2] if centroids.shape[1] > 2 else [0] * len(centroids), # Check if there are 3 centroids, if not set z to 0
            mode='markers',
            marker=dict(
                size=8,
                color='red',  # set color of the centroids
                opacity=0.8
            ),
            name='Centroids'
        ))

        # Update the layout
        fig.update_layout(
```

```python
            title=dict(
                text = title,
                y = 0.95
                ),
            height = 750,
            scene=dict(
                xaxis_title=labels[0],
                yaxis_title=labels[1],
                zaxis_title=labels[2]
            ),
            margin=dict(l=0, r=0, b=0, t=0)  # Adjust margins to fit labels, if necessary
        )




        fig.show()
        if 'dispatched' in df.columns:
            print("RATIO OF DISPATCHED TO NON-DISPATCHED BIDS IN EACH CLUSTER")
            dispatched_ratio(df)

            print("\n ________________________________________________________________________________________ \n")

        print(f"WEIGHTED MEAN AND VARIANCE OF {features[0].upper()}, {features[1].upper()} AND {features[2].upper()} FOR EACH CLUSTER")
        cluster_means_variances(df, features)

        print("\n ________________________________________________________________________________________ \n")

        print("CLUSTER SIZES")
        cluster_sizes(df)

        print("\n ________________________________________________________________________________________ \n")

        print("CENTROID VALUES FOR EACH CLUSTER:")
        cluster_centroids(df, features)

    if len(features) > 3:
        print('Too many features to plot, please refer to the cluster table for more information.')

        print("\n ________________________________________________________________________________________ \n")

        if 'dispatched' in df.columns:
            print("RATIO OF DISPATCHED TO NON-DISPATCHED BIDS IN EACH CLUSTER")
            dispatched_ratio(df)

            print("\n ________________________________________________________________________________________ \n")

        print(f"WEIGHTED MEAN AND VARIANCE OF FEATURES FOR EACH CLUSTER")
        cluster_means_variances(df, features)

        print("\n ________________________________________________________________________________________ \n")

        print("CLUSTER SIZES")
        cluster_sizes(df)

        print("\n ________________________________________________________________________________________ \n")

        print("CENTROID VALUES FOR EACH CLUSTER:")
        cluster_centroids(df, features)



#plot a scatter plot of the data before clustering **(FOR DEMONSTRATION PURPOSES ONLY)**
def scatter_plot(df, features, title):
    fig = px.scatter(df, x=features[0], y=features[1], hover_data=['asset_ID'], title=title)
    fig.show()



# function to plot the current clusters and centroids **(FOR DEMONSTRATION PURPOSES ONLY)**
def plot_clusters(df, centroids, labels, title):
    plt.figure(figsize=(8, 6))
    plt.scatter(df.iloc[:, 0], df.iloc[:, 1], c=labels, cmap='viridis', marker='o', s=30, alpha=0.6)
    plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', s=100, label='Centroids')
    plt.title(title)
    plt.xlabel('Block Price ($)')
    plt.ylabel('Block Size (MW)')
    plt.legend()
    plt.show()

# function to perform kmeans clustering iteratively **(FOR DEMONSTRATION PURPOSES ONLY)**
def kmeans_cluster_iter(df, features, nclusters, n_init=10):
    centroids = None
    for i in range(n_init):
        # Selecting specific features for clustering
        X = df[features].values
        kmeans = KMeans(n_clusters=nclusters, init='k-means++' if centroids is None else centroids, max_iter=1, n_init=1, random_state=i)
        kmeans.fit(X)

        if i == 0:
            # Initial centroid placement and cluster distinctions
            plot_clusters(pd.DataFrame(X, columns=features), kmeans.cluster_centers_, kmeans.labels_, 'Initial Cluster Distinctions')

        centroids = kmeans.cluster_centers_ # Update centroids for next initialization
        if i == 1:
            # 2nd centroid placement and cluster distinctions
            plot_clusters(pd.DataFrame(X, columns=features), centroids, kmeans.labels_, '2nd Cluster Distinctions')
```

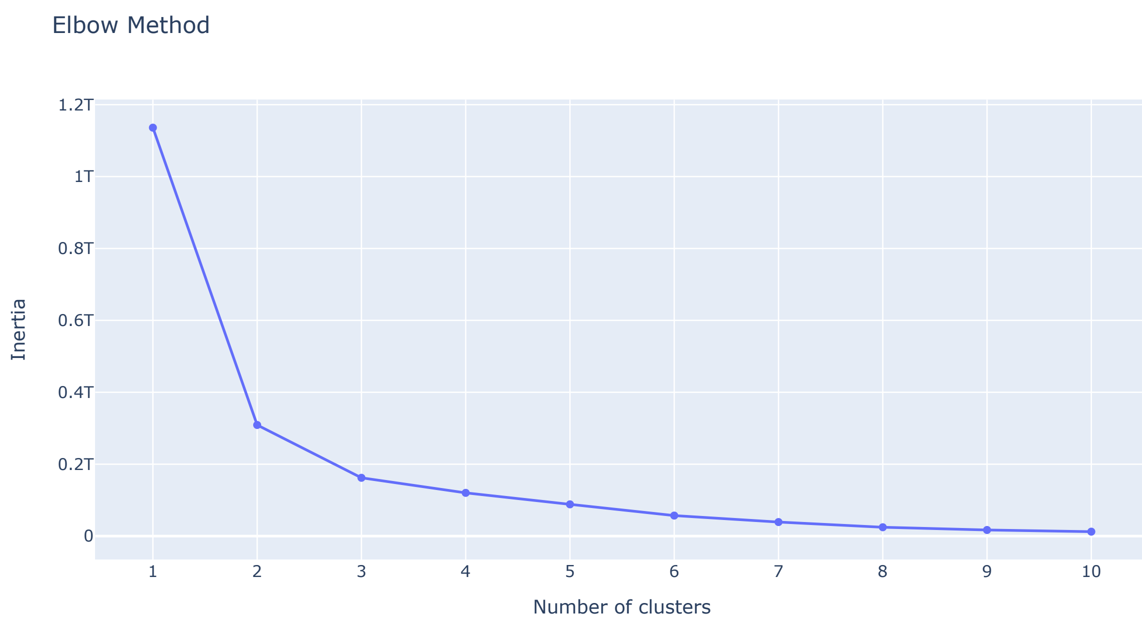
```
        centroids = kmeans.cluster_centers_ # Update centroids for next initialization


    # Final clusters and distinctions
    kmeans.fit(X) # Final fit to get the end state
    plot_clusters(pd.DataFrame(X, columns=features), kmeans.cluster_centers_, kmeans.labels_, 'Final Clusters and Distinctions')



# using daily_master_df, create a new data frame that aggregates each generators data for the given features in aggregate_features
def aggregate_data(df):
    # create a new dataframe to store the aggregated data
    aggregated_data = pd.DataFrame()
    #aggregated_data["zero_offer_percent"] = 1000
    # loop through the generators
    for generator in df['asset_ID'].unique():
        # create a dictionary to store the aggregated data for the generator
        generator_data = {'asset_ID': generator}
        # loop through the features
        for feature in df.columns:
            if feature == 'dispatched' or feature == 'flexible':
                # count the number of observations that equal 1 for the feature
                generator_data[f'count_{feature}'] = df[(df['asset_ID'] == generator) & (df[feature] == 1)].shape[0]
            elif feature == 'asset_ID' or feature == 'begin_dateTime_mpt' or feature == 'Unnamed: 0' or feature == 'offer_control':
                # skip these features
                continue
            else:
                # calculate the sum of the feature for the generator
                generator_data[f'sum_{feature}'] = df[df['asset_ID'] == generator][feature].sum()



        # append the generator data to the aggregated data dataframe
        aggregated_data = aggregated_data._append(generator_data, ignore_index=True)

    aggregated_data["zero_offer_percent"] = 0.01
    # create a new column in the aggregated data dataframe that contains the percentage of total capacity offered (sum_block_size) that was 
    for generator in df["asset_ID"].unique():
        aggregated_data.loc[aggregated_data["asset_ID"] == generator, "zero_offer_percent"] = df[(df["asset_ID"] == generator) & (df["block_p


    # create a new column in the aggregated data dataframe that contains the percentage of total capacity offered (sum_block_size) that was 
    for generator in df["asset_ID"].unique():
        aggregated_data.loc[aggregated_data["asset_ID"] == generator, "sum_dispatched_block_price"] = df[(df["asset_ID"] == generator) & (df

    # create a new column in the aggregated data dataframe that contains the percentage of total capacity offered (sum_block_size) that was 
    for generator in df["asset_ID"].unique():
        aggregated_data.loc[aggregated_data["asset_ID"] == generator, "sum_dispatched_block_size"] = df[(df["asset_ID"] == generator) & (df['


    # return the aggregated data dataframe
    return aggregated_data
```

Run the Cluster Function and Display Cluster Statistics

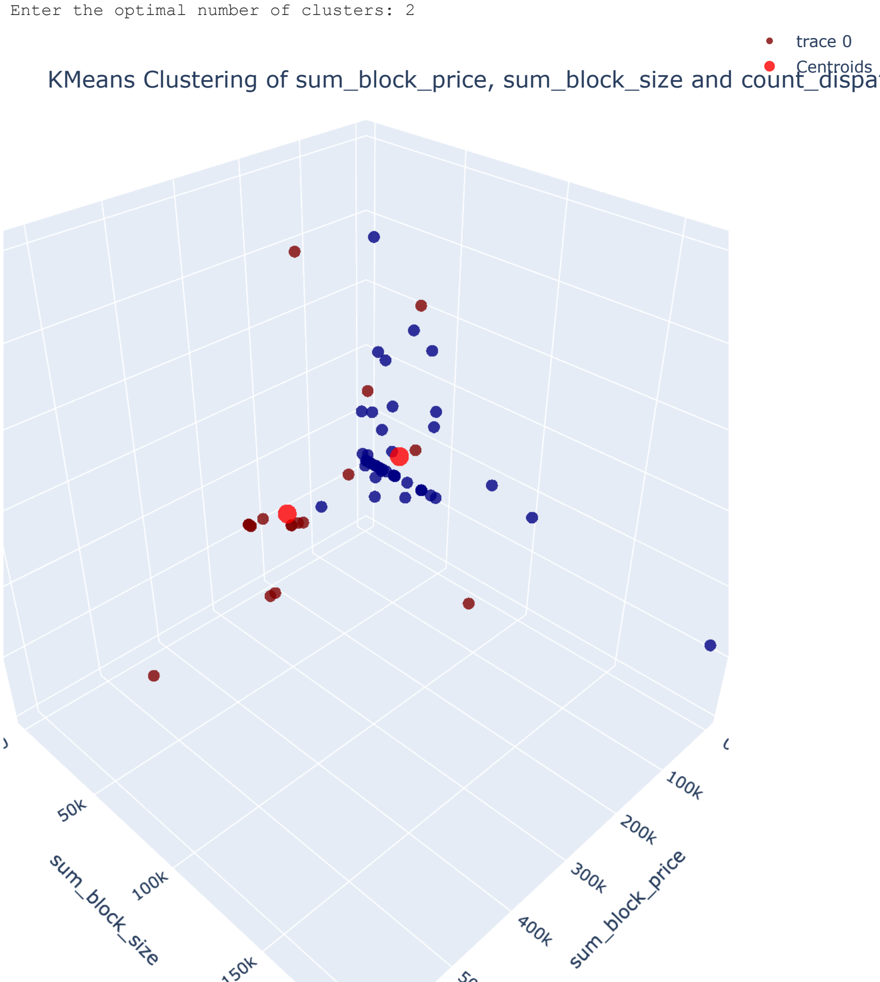
In [4]:
```
aggregate_choice = input('Would you like to aggregate the data before clustering? (Yes/No): ').lower()




if aggregate_choice == 'yes':
    aggregate_features_input = input('Enter the features you would like to cluster by separated by a space: ').split()
    aggregate_data_df = aggregate_data(daily_master_df)
    # add sum prefix to columns in aggregate_features
    for i in range(len(aggregate_features_input)):
        if aggregate_features_input[i] != 'dispatched' and aggregate_features_input[i] != 'flexible' and aggregate_features_input[i] != 'zerc
            aggregate_features_input[i] = 'sum_' + aggregate_features_input[i]
        elif aggregate_features_input[i] == 'dispatched' or aggregate_features_input[i] == 'flexible':
            aggregate_features_input[i] = 'count_' + aggregate_features_input[i]
        elif aggregate_features_input[i] == 'zero_offer_percent' or aggregate_features_input[i] == 'sum_dispatched_block_price' or aggregate_
            continue

    # perform the elbow method
    elbow_method(aggregate_data_df, aggregate_features_input)
    # prompt the user for the optimal number of clusters as determined by the elbow method plot
    optimal_clusters = int(input('Enter the optimal number of clusters: '))
    # perform the kmeans clustering on the aggregated data
    if len(aggregate_features_input) == 1:
        print("Error! Cannot perform clustering with only one feature. Please enter at least two features.")
    if len(aggregate_features_input) == 2:
        kmeans_cluster(aggregate_data_df, aggregate_features_input, optimal_clusters, aggregate_features_input, f'KMeans Clustering of {aggre
        cluster_assets(aggregate_data_df)
    if len(aggregate_features_input) == 3:
        kmeans_cluster(aggregate_data_df, aggregate_features_input, optimal_clusters, aggregate_features_input, f'KMeans Clustering of {aggre
        cluster_assets(aggregate_data_df)
    if len(aggregate_features_input) > 3:
        kmeans_cluster(aggregate_data_df, aggregate_features_input, optimal_clusters, aggregate_features_input, f'KMeans Clustering of Select
        cluster_assets(aggregate_data_df)

elif aggregate_choice == 'no':
    # prompt the user for the features they would like to cluster by
    features_input = input('Enter the features you would like to cluster by, separated by a space: ').split()

    # perform the elbow method
    elbow_method(daily_master_df, features_input)
```

```
    # prompt the user for the optimal number of clusters as determined by the elbow method plot
    optimal_clusters = int(input('Enter the optimal number of clusters: '))
    if len(features_input) == 1:
        print("Error! Cannot perform clustering with only one feature. Please enter at least two features.")

    if len(features_input) == 2:
        kmeans_cluster(daily_master_df, features_input, optimal_clusters, features_input, f'KMeans Clustering of {features_input[0]} and {fea

    if len(features_input) == 3:
        kmeans_cluster(daily_master_df, features_input, optimal_clusters, features_input, f'KMeans Clustering of {features_input[0]}, {featu

    if len(features_input) > 3:
        kmeans_cluster(daily_master_df, features_input, optimal_clusters, features_input, f'KMeans Clustering of Selected Features')

else:
    print('Invalid input. Please enter either "Yes" or "No".')
```

```
Would you like to aggregate the data before clustering? (Yes/No): yes
Enter the features you would like to cluster by separated by a space: block_price block_size dispatched
```




```
Enter the optimal number of clusters: 2
```



WEIGHTED MEAN AND VARIANCE OF SUM_BLOCK_PRICE, SUM_BLOCK_SIZE AND COUNT_DISPATCHED FOR EACH CLUSTER

	cluster	mean_sum_block_price	var_sum_block_price	mean_sum_block_size	var_sum_block_size	mean_count_dispatched	var_count_dispatched
0	1	243628.904583	8.214902e+09	29740.166667	1.758234e+09	367.083333	104349.036232
1	0	13787.784889	6.939473e+08	27063.977778	1.116644e+09	339.288889	35260.573737

CLUSTER SIZES

	cluster	size
0	0	45
0	1	24

CENTROID VALUES FOR EACH CLUSTER:

	centroid	sum_block_price	sum_block_size	count_dispatched
0	[1]	243628.904583	29740.166667	367.083333
1	[0]	13787.784889	27063.977778	339.288889

Cluster 1

	asset_ID
0	BHL1
1	COD1
2	GEN5
3	ME04
4	BFD1
5	ALP1
6	ME03
7	ME02
8	ALP2
9	SDH1
10	CMH1
11	BR5
12	SCR6
13	CRS3
14	VVW1
15	VVW2
16	CRS2
17	HRV1
18	SH2
19	SD6
20	KH2
21	SH1
65	NPC3
66	NPC2

Cluster 0

	asset_ID
22	EC01
23	DOWG
24	ALS1
25	NPP1
26	SCL1
27	NAT1
28	ANC1
29	ENC3
30	KH3
31	GEN6
32	SET1
33	JOF1
34	FH1
35	MKR1

36	TC01
37	DRW1
38	MUL1
39	UOC1
40	TLM2
41	WCD1
42	BCR2
43	SCR1
44	PH1
45	CRS1
46	HMT1
47	RL1
48	HRT1
49	RB5
50	CL01
51	FNG1
52	BCRK
53	EC04
54	PR1
55	TC02
56	MEG1
57	APS1
58	BR4
59	IOR2
60	IOR1
61	NX02
62	MKRC
63	SCR5
64	EGC1
67	NX01
68	CAL1

In [ ]: