Customer Segmentation with K-Means Clustering

Customer segmentation will be applied to an e-commerce customer database using K-means clustering from scikit-learn.

The provided customers database is visualized as part of a case study. This project is taking the case study one step further with the following motive:

Can this customer database be grouped to develop customized relationships?

To answer this question 3 features will be created and used:

- products ordered
- average return rate
- total spending

Dataset represents real customers & orders data between November 2018 - April 2019 and it is pseudonymized for confidentiality.

Imports

In [1]:
```
# data wrangling
import pandas as pd
import numpy as np

# visualization
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# for data preprocessing and clustering
from sklearn.cluster import KMeans

%matplotlib inline
# to include graphs inline within the frontends next to code

%config InlineBackend.figure_format='retina'
#to enable retina (high resolution) plots

pd.options.mode.chained_assignment = None
# to bypass warnings in various dataframe assignments
```

Investigate data

In [2]:
```
# load data into a dataframe
customers_orders = pd.read_csv("Orders - Analysis Task.csv")
```

In [3]:
```
# first rows of the dataset
customers_orders.head(15)
```

Out[3]:

	product_title	product_type	variant_title	variant_sku	variant_id	customer_id	order_id

	product_title	product_type	variant_title	variant_sku	variant_id	customer_id	order_id
0	DPR	DPR	100	AD-982-708-895-F-6C894FB	52039657	1312378	83290718932496
1	RJF	Product P	28 / A / MTM	83-490-E49-8C8-8-3B100BC	56914686	3715657	36253792848113
2	CLH	Product B	32 / B / FtO	68-ECA-BC7-3B2-A-E73DE1B	24064862	9533448	73094559597229
3	NMA	Product F	40 / B / FtO	6C-1F1-226-1B3-2-3542B41	43823868	4121004	53616575668264
4	NMA	Product F	40 / B / FtO	6C-1F1-226-1B3-2-3542B41	43823868	4121004	29263220319421
5	OTH	Product F	40 / B / FtO	53-5CA-7CF-8F5-9-28CB78B	43823868	4121004	53616575668264
6	OTH	Product F	40 / B / FtO	53-5CA-7CF-8F5-9-28CB78B	43823868	4121004	29263220319421
7	NMA	Product F	40 / B / FtO	6C-1F1-226-1B3-2-3542B41	43823868	4121004	13666410519728
8	OTH	Product F	40 / C / FtO	8B-2C5-548-6C6-E-B5EECBC	43823868	4121004	80657249973427
9	DPR	DPR	100	AD-982-708-895-F-6C894FB	52039657	1312378	48128811961800
10	DPR	DPR	100	AD-982-708-895-F-6C894FB	52039657	1312378	16219792823741
11	DPR	DPR	100	AD-982-708-895-F-6C894FB	52039657	1312378	61841856022929
12	LQS	Product C	33 / A / FtO	C7-247-274-5DF-F-3B09882	26922227	4029832	31862450581156
13	CLH	Product B	33 / A / FtO	E0-AFF-619-205-D-6AA15A5	24064862	2864696	31862450581156
14	RJF	Product P	28 / A / MTM	83-490-E49-8C8-8-3B100BC	56914686	3715657	14741469246913

In [4]:

```
# first glance of customers_orders data
customers_orders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 70052 entries, 0 to 70051
Data columns (total 17 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   product_title           70052 non-null  object
 1   product_type            70052 non-null  object
 2   variant_title           70052 non-null  object
 3   variant_sku             70052 non-null  object
 4   variant_id              70052 non-null  int64
 5   customer_id             70052 non-null  int64
 6   order_id                70052 non-null  int64
 7   day                     70052 non-null  object
 8   net_quantity            70052 non-null  int64
 9   gross_sales             70052 non-null  float64
 10  discounts               70052 non-null  float64
 11  returns                 70052 non-null  float64
 12  net_sales               70052 non-null  float64
 13  taxes                   70052 non-null  float64
 14  total_sales             70052 non-null  float64
 15  returned_item_quantity  70052 non-null  int64
 16  ordered_item_quantity   70052 non-null  int64
dtypes: float64(6), int64(6), object(5)
memory usage: 9.1+ MB
```

In [5]:

```
# descriptive statistics of the non-object columns
customers_orders.describe()
```

Out[5]:

	variant_id	customer_id	order_id	net_quantity	gross_sales	discounts	r
count	7.005200e+04	7.005200e+04	7.005200e+04	70052.000000	70052.000000	70052.000000	70052.(
mean	2.442320e+11	6.013091e+11	5.506075e+13	0.701179	61.776302	-4.949904	-10.2
std	4.255079e+12	6.223201e+12	2.587640e+13	0.739497	31.800689	7.769972	25.1
min	1.001447e+07	1.000661e+06	1.000657e+13	-3.000000	0.000000	-200.000000	-237.5
25%	2.692223e+07	3.295695e+06	3.270317e+13	1.000000	51.670000	-8.340000	0.(
50%	4.494514e+07	5.566107e+06	5.522207e+13	1.000000	74.170000	0.000000	0.(
75%	7.743106e+07	7.815352e+06	7.736876e+13	1.000000	79.170000	0.000000	0.(
max	8.422212e+13	9.977409e+13	9.999554e+13	6.000000	445.000000	0.000000	0.(

There were significant number of rows whose `ordered_item_quantity` is 0 and `net_quantity` is less than 0, which means they are not ordered/sold at all; but the fact that they have returns requires investigation.

In [6]:

```
print("Number of rows that net quantity is negative:",
      customers_orders[customers_orders.net_quantity < 0].shape[0])
```

```
Number of rows that net quantity is negative: 10715
```

These rows will be excluded from the orders dataset for the project.

In [7]:

```
# exclude not sold/ordered SKUs from the dataset
customers_orders = customers_orders[
    customers_orders["ordered_item_quantity"] > 0]
```

1. Products ordered

It is the count of the products ordered in product_type column by a customer.

Create functions to identify customers who order multiple products

In [8]:
```
def encode_column(column):
    if column > 0:
        return 1
    if column <= 0:
        return 0


def aggregate_by_ordered_quantity(dataframe, column_list):
    '''this function:
    1. aggregates a given dataframe by column list,
    as a result creates a aggregated dataframe by counting the ordered item quantiti

    2. adds number_of_X ordered where X is the second element in the column_list
    to the aggregated dataframe by encoding ordered items into 1

    3. creates final dataframe containing information about
    how many of X are ordered, based on the first element passed in the column list'

    aggregated_dataframe = (dataframe
                            .groupby(column_list)
                            .ordered_item_quantity.count()
                            .reset_index())

    aggregated_dataframe["products_ordered"] = (aggregated_dataframe
                                                .ordered_item_quantity
                                                .apply(encode_column))

    final_dataframe = (aggregated_dataframe
                       .groupby(column_list[0])
                       .products_ordered.sum() # aligned with the added column name
                       .reset_index())

    return final_dataframe
```

In [9]:
```
# apply functions to customers_orders
customers = aggregate_by_ordered_quantity(customers_orders, ["customer_id", "product
```

In [10]:
```
print(customers.head())
```

```
   customer_id  products_ordered
0      1000661                 1
1      1001914                 1
2      1002167                 3
3      1002387                 1
4      1002419                 2
```

2. Average Return Rate

It is the ratio of returned item quantity and ordered item quantity. This ratio is first calculated per order and then averaged for all orders of a customer.

In [11]:
```
# aggregate data per customer_id and order_id,
# to see ordered item sum and returned item sum
ordered_sum_by_customer_order = (customers_orders
```

```
                                    .groupby(["customer_id", "order_id"])
                                    .ordered_item_quantity.sum()
                                    .reset_index())

returned_sum_by_customer_order = (customers_orders
                                    .groupby(["customer_id", "order_id"])
                                    .returned_item_quantity.sum()
                                    .reset_index())

# merge two dataframes to be able to calculate unit return rate
ordered_returned_sums = pd.merge(ordered_sum_by_customer_order, returned_sum_by_cust
```

In [12]:

```
# calculate unit return rate per order and customer
ordered_returned_sums["average_return_rate"] = (-1 *
                                                ordered_returned_sums["returned_item_qu
                                                ordered_returned_sums["ordered_item_qua
```

In [13]:

```
ordered_returned_sums.head()
```

Out[13]:

	customer_id	order_id	ordered_item_quantity	returned_item_quantity	average_return_rate
0	1000661	99119989117212	3	0	0.0
1	1001914	79758569034715	1	0	0.0
2	1002167	38156088848638	1	0	0.0
3	1002167	57440147820257	1	0	0.0
4	1002167	58825523953710	1	0	0.0

In [14]:

```
# take average of the unit return rate for all orders of a customer
customer_return_rate = (ordered_returned_sums
                        .groupby("customer_id")
                        .average_return_rate
                        .mean()
                        .reset_index())
```

In [15]:

```
return_rates = pd.DataFrame(customer_return_rate["average_return_rate"]
                            .value_counts()
                            .reset_index())

return_rates.rename(columns=
                    {"index": "average return rate",
                     "average_return_rate": "count of unit return rate"},
                    inplace=True)

return_rates.sort_values(by="average return rate")
```

Out[15]:

	average return rate	count of unit return rate
0	0.000000	24823
8	0.013889	1
9	0.066667	1
11	0.083333	1

	average return rate	count of unit return rate
10	0.125000	1
6	0.166667	2
5	0.200000	2
4	0.250000	5
2	0.333333	13
12	0.400000	1
3	0.500000	9
7	0.666667	2
1	1.000000	13

In [16]:
```
# add average_return_rate to customers dataframe
customers = pd.merge(customers,
                     customer_return_rate,
                     on="customer_id")
```

3. Total spending

Total spending is the aggregated sum of total sales value which is the amount after the taxes and returns.

In [17]:
```
# aggreagate total sales per customer id
customer_total_spending = (customers_orders
                           .groupby("customer_id")
                           .total_sales
                           .sum()
                           .reset_index())

customer_total_spending.rename(columns = {"total_sales" : "total_spending"},
                               inplace = True)
```

Create features data frame

In [18]:
```
# add total sales to customers dataframe
customers = customers.merge(customer_total_spending,
                            on="customer_id")
```

In [19]:
```
print("The number of customers from the existing customer base:", customers.shape[0]
```

```
The number of customers from the existing customer base: 24874
```

In [20]:
```
# drop id column since it is not a feature
customers.drop(columns="customer_id",
               inplace=True)
```

In [21]:
```
customers.head()
```

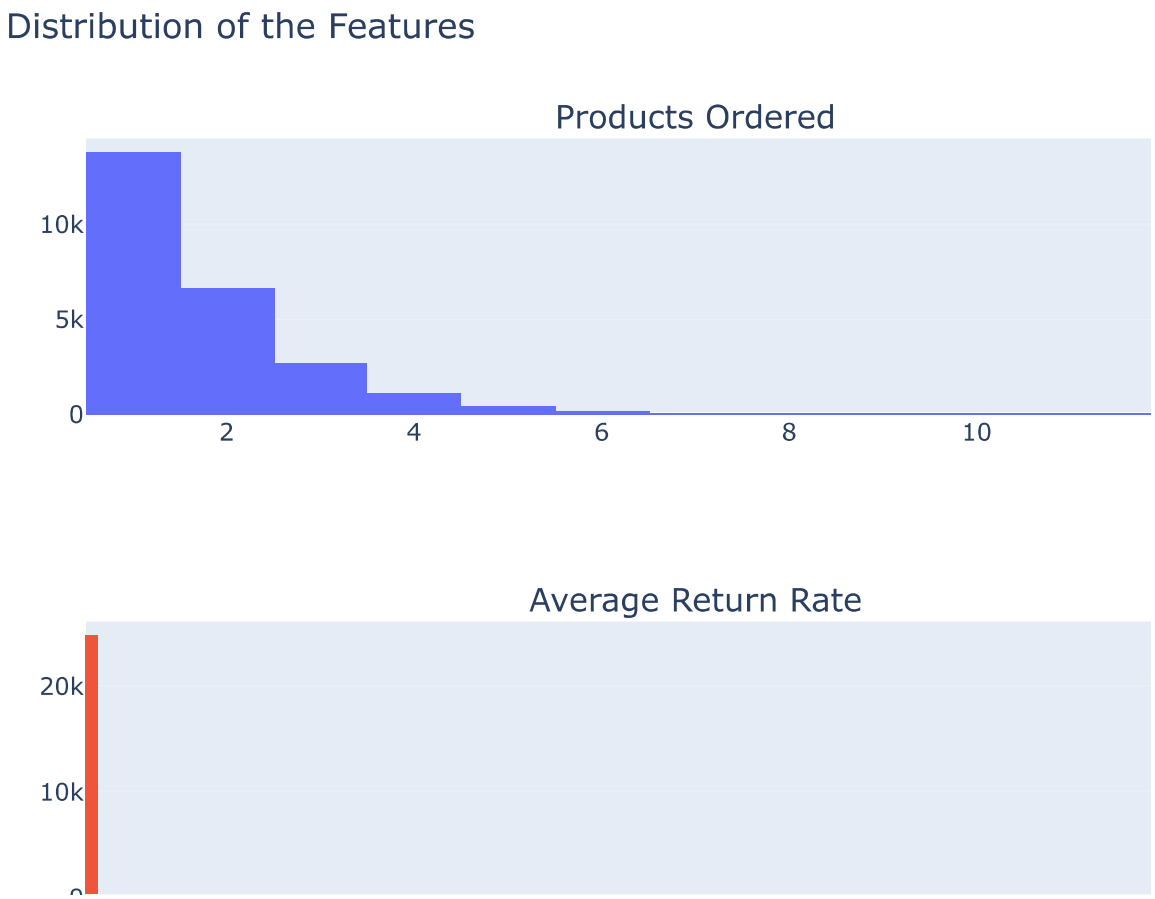
Out[21]:

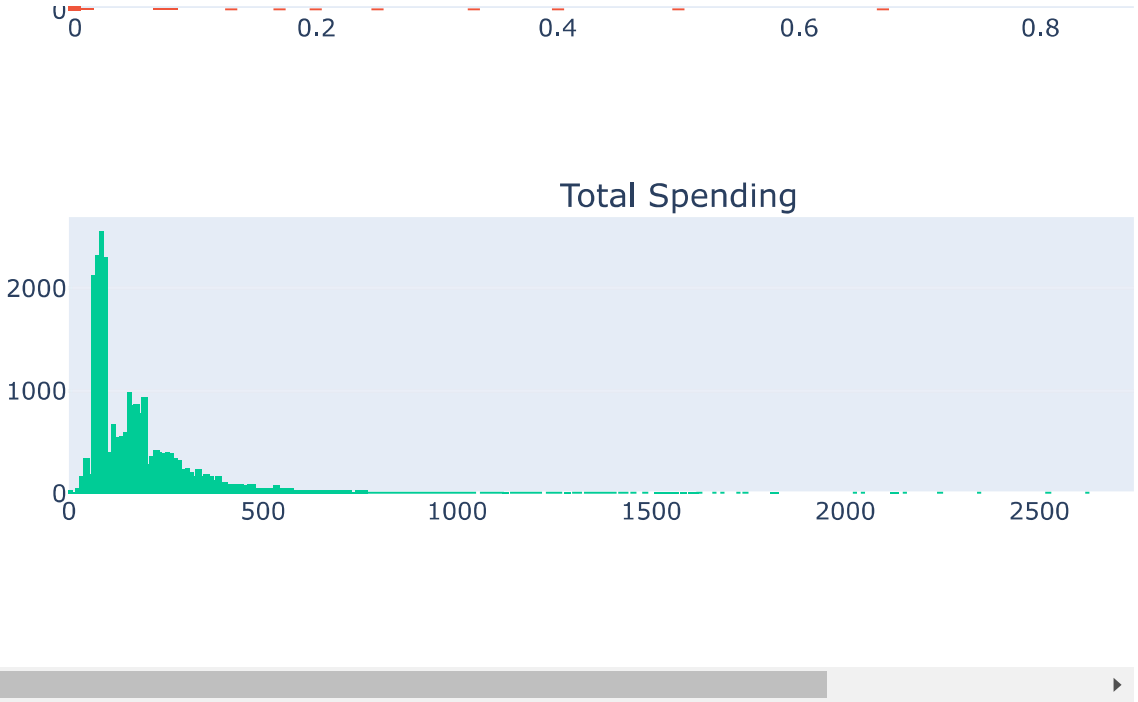
	products_ordered	average_return_rate	total_spending
0	1	0.0	260.0
1	1	0.0	79.2
2	3	0.0	234.2
3	1	0.0	89.0
4	2	0.0	103.0

Visualize features

In [22]:

```
fig = make_subplots(rows=3, cols=1,
                    subplot_titles=("Products Ordered",
                                    "Average Return Rate",
                                    "Total Spending"))

fig.append_trace(go.Histogram(x=customers.products_ordered),
                 row=1, col=1)

fig.append_trace(go.Histogram(x=customers.average_return_rate),
                 row=2, col=1)

fig.append_trace(go.Histogram(x=customers.total_spending),
                 row=3, col=1)

fig.update_layout(height=800, width=800,
                  title_text="Distribution of the Features")

fig.show()
```





Scale Features: Log Transformation

In [23]:
```
def apply_log1p_transformation(dataframe, column):
    '''This function takes a dataframe and a column in the string format
    then applies numpy log1p transformation to the column
    as a result returns log1p applied pandas series'''

    dataframe["log_" + column] = np.log1p(dataframe[column])
    return dataframe["log_" + column]
```

1. Products ordered

In [24]:
```
apply_log1p_transformation(customers, "products_ordered")
```

Out[24]:
```
0        0.693147
1        0.693147
2        1.386294
3        0.693147
4        1.098612
           ...
24869    1.098612
24870    1.098612
24871    0.693147
24872    1.098612
24873    0.693147
Name: log_products_ordered, Length: 24874, dtype: float64
```

2. Average return rate

In [25]:
```
apply_log1p_transformation(customers, "average_return_rate")
```

Out[25]:
```
0        0.0
1        0.0
2        0.0
3        0.0
4        0.0
          ...
24869    0.0
```

```
24870    0.0
24871    0.0
24872    0.0
24873    0.0
Name: log_average_return_rate, Length: 24874, dtype: float64
```

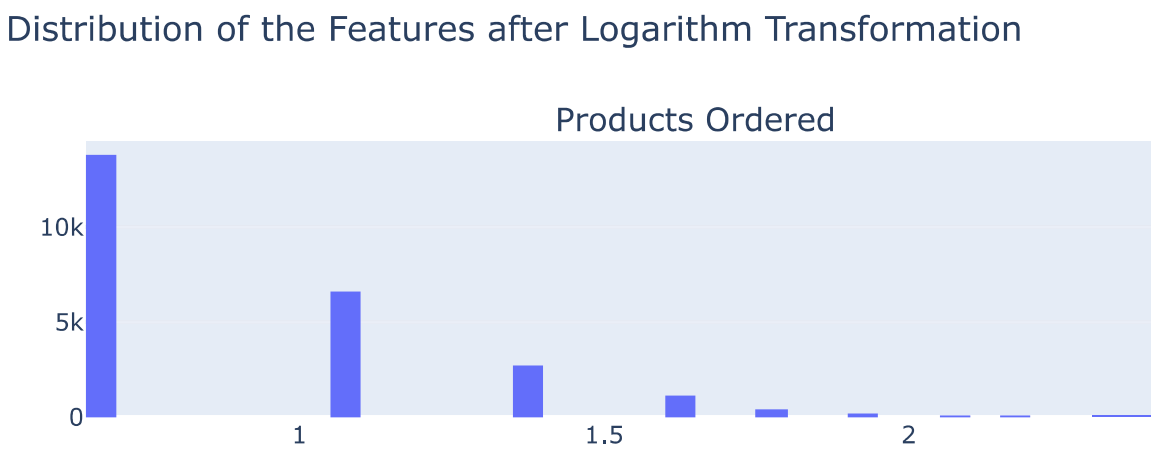
3. Total spending

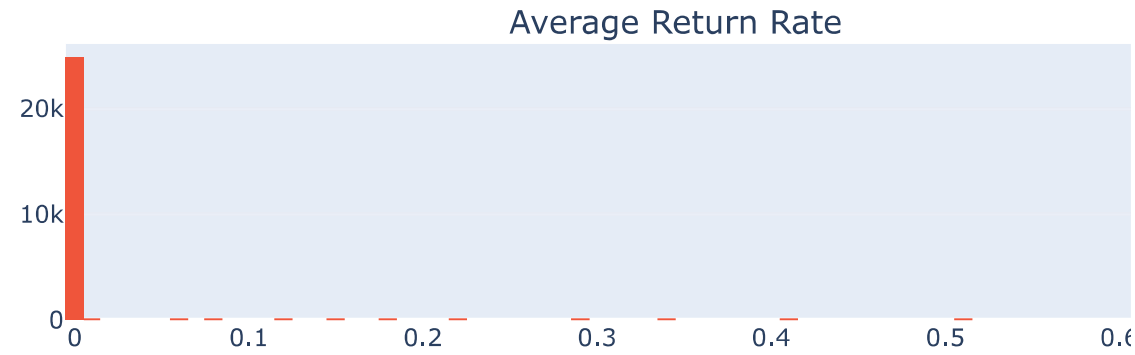
In [26]:
```
apply_log1p_transformation(customers, "total_spending")
```

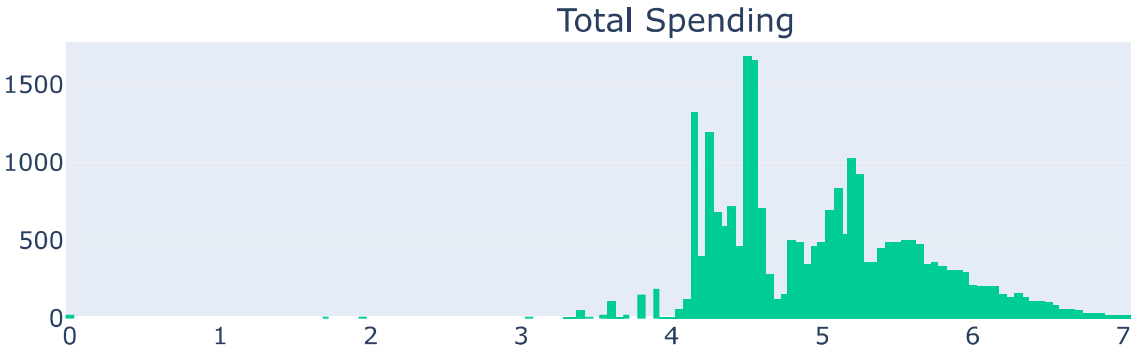
Out[26]:
```
0        5.564520
1        4.384524
2        5.460436
3        4.499810
4        4.644391
           ...
24869    5.560682
24870    5.495117
24871    4.499810
24872    5.590987
24873    4.174387
Name: log_total_spending, Length: 24874, dtype: float64
```

Visualize log transformation applied features

In [27]:
```
fig = make_subplots(rows=3, cols=1,
                    subplot_titles=("Products Ordered",
                                    "Average Return Rate",
                                    "Total Spending"))

fig.append_trace(go.Histogram(x=customers.log_products_ordered),
                 row=1, col=1)

fig.append_trace(go.Histogram(x=customers.log_average_return_rate),
                 row=2, col=1)

fig.append_trace(go.Histogram(x=customers.log_total_spending),
                 row=3, col=1)

fig.update_layout(height=800, width=800,
                  title_text="Distribution of the Features after Logarithm Transform

fig.show()
```







In [28]:
```
customers.head()
```

Out[28]:

	products_ordered	average_return_rate	total_spending	log_products_ordered	log_average_return_r
0	1	0.0	260.0	0.693147	
1	1	0.0	79.2	0.693147	
2	3	0.0	234.2	1.386294	
3	1	0.0	89.0	0.693147	
4	2	0.0	103.0	1.098612	

In [29]:
```
# features we are going to use as an input to the model
customers.iloc[:,3:]
```

Out[29]:

	log_products_ordered	log_average_return_rate	log_total_spending
0	0.693147	0.0	5.564520
1	0.693147	0.0	4.384524
2	1.386294	0.0	5.460436
3	0.693147	0.0	4.499810
4	1.098612	0.0	4.644391
...	...	...	...

	log_products_ordered	log_average_return_rate	log_total_spending
24869	1.098612	0.0	5.560682
24870	1.098612	0.0	5.495117
24871	0.693147	0.0	4.499810
24872	1.098612	0.0	5.590987
24873	0.693147	0.0	4.174387

24874 rows × 3 columns

Create K-means model

In [30]:
```
# create initial K-means model
kmeans_model = KMeans(init='k-means++',
                      max_iter=500,
                      random_state=42)
```

In [31]:
```
kmeans_model.fit(customers.iloc[:,3:])

# print the sum of distances from all examples to the center of the cluster
print("within-cluster sum-of-squares (inertia) of the model is:", kmeans_model.inert
```

```
within-cluster sum-of-squares (inertia) of the model is: 1067.543498818075
```

Hyperparameter tuning: Find optimal number of clusters

In [32]:
```
def make_list_of_K(K, dataframe):
    '''inputs: K as integer and dataframe
    apply k-means clustering to dataframe
    and make a list of inertia values against 1 to K (inclusive)
    return the inertia values list
    '''

    cluster_values = list(range(1, K+1))
    inertia_values=[]

    for c in cluster_values:
        model = KMeans(
            n_clusters = c,
            init='k-means++',
            max_iter=500,
            random_state=42)
        model.fit(dataframe)
        inertia_values.append(model.inertia_)

    return inertia_values
```

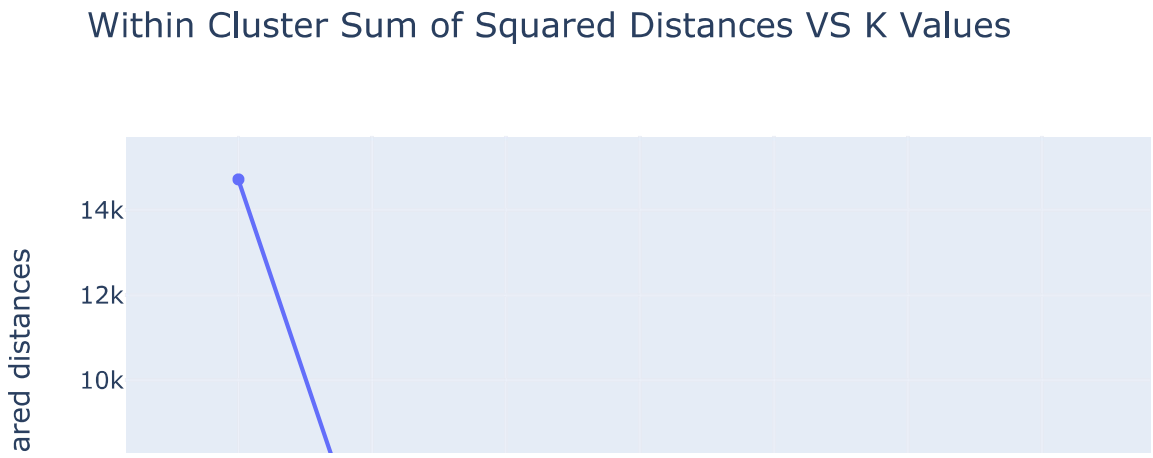
Visualize different K and models

In [33]:
```
# save inertia values in a dataframe for k values between 1 to 15
results = make_list_of_K(15, customers.iloc[:, 3:])
```

```
k_values_distances = pd.DataFrame({"clusters": list(range(1, 16)),
                                   "within cluster sum of squared distances": result
```

In [34]:

```
# visualization for the selection of number of segments
fig = go.Figure()

fig.add_trace(go.Scatter(x=k_values_distances["clusters"],
                         y=k_values_distances["within cluster sum of squared distanc
                         mode='lines+markers'))

fig.update_layout(xaxis = dict(
        tickmode = 'linear',
        tick0 = 1,
        dtick = 1),
                  title_text="Within Cluster Sum of Squared Distances VS K Values",
                  xaxis_title="K values",
                  yaxis_title="Cluster sum of squared distances")

fig.show()
```



Update K-Means Clustering

In [35]:

```
# create clustering model with optimal k=4
updated_kmeans_model = KMeans(n_clusters = 4,
                              init='k-means++',
                              max_iter=500,
                              random_state=42)
```

```
updated_kmeans_model.fit_predict(customers.iloc[:,3:])
```

Out[35]:
```
array([3, 1, 3, ..., 1, 3, 1])
```

Add cluster centers to the visualization

In [36]:
```
# create cluster centers and actual data arrays
cluster_centers = updated_kmeans_model.cluster_centers_
actual_data = np.expm1(cluster_centers)
add_points = np.append(actual_data, cluster_centers, axis=1)
add_points
```

Out[36]:
```
array([[3.94249145e+00, 5.53287649e-04, 5.79614576e+02, 1.59786955e+00,
        5.53134642e-04, 6.36408716e+00],
       [1.01496335e+00, 1.15284613e-03, 7.65510085e+01, 7.00601007e-01,
        1.15218211e-03, 4.35093589e+00],
       [1.52712228e+00, 5.47294198e-04, 1.59893528e+02, 9.27081218e-01,
        5.47144488e-04, 5.08074283e+00],
       [2.39675970e+00, 5.21304945e-04, 2.83647970e+02, 1.22282195e+00,
        5.21169113e-04, 5.65125322e+00]])
```

In [37]:
```
# add labels to customers dataframe and add_points array
add_points = np.append(add_points, [[0], [1], [2], [3]], axis=1)
customers["clusters"] = updated_kmeans_model.labels_
```

In [38]:
```
# create centers dataframe from add_points
centers_df = pd.DataFrame(data=add_points, columns=["products_ordered",
                                                    "average_return_rate",
                                                    "total_spending",
                                                    "log_products_ordered",
                                                    "log_average_return_rate",
                                                    "log_total_spending",
                                                    "clusters"])
centers_df.head()
```

Out[38]:

	products_ordered	average_return_rate	total_spending	log_products_ordered	log_average_return_r
0	3.942491	0.000553	579.614576	1.597870	0.000
1	1.014963	0.001153	76.551009	0.700601	0.001
2	1.527122	0.000547	159.893528	0.927081	0.000
3	2.396760	0.000521	283.647970	1.222822	0.000

In [39]:
```
# align cluster centers of centers_df and customers
centers_df["clusters"] = centers_df["clusters"].astype("int")
```

In [40]:
```
centers_df.head()
```

Out[40]:

	products_ordered	average_return_rate	total_spending	log_products_ordered	log_average_return_r
0	3.942491	0.000553	579.614576	1.597870	0.000
1	1.014963	0.001153	76.551009	0.700601	0.001

	products_ordered	average_return_rate	total_spending	log_products_ordered	log_average_return_r
2	1.527122	0.000547	159.893528	0.927081	0.000
3	2.396760	0.000521	283.647970	1.222822	0.000

In [41]:
```
customers.head()
```

Out[41]:

	products_ordered	average_return_rate	total_spending	log_products_ordered	log_average_return_r
0	1	0.0	260.0	0.693147	
1	1	0.0	79.2	0.693147	
2	3	0.0	234.2	1.386294	
3	1	0.0	89.0	0.693147	
4	2	0.0	103.0	1.098612	

In [42]:
```
# differentiate between data points and cluster centers
customers["is_center"] = 0
centers_df["is_center"] = 1

# add dataframes together
customers = customers.append(centers_df, ignore_index=True)
```

In [43]:
```
customers.tail()
```

Out[43]:

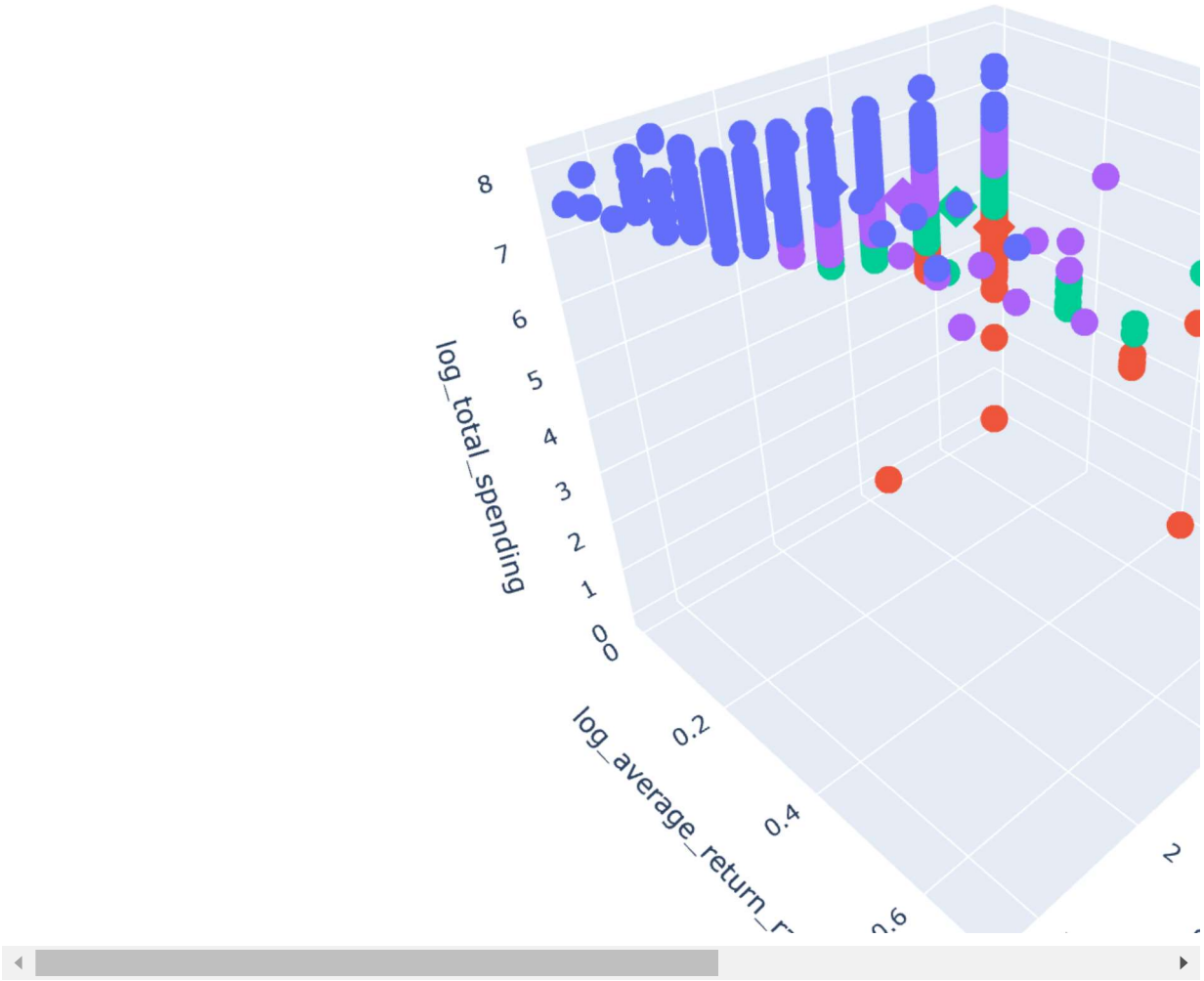
	products_ordered	average_return_rate	total_spending	log_products_ordered	log_average_ret
24873	1.000000	0.000000	64.000000	0.693147	
24874	3.942491	0.000553	579.614576	1.597870	
24875	1.014963	0.001153	76.551009	0.700601	
24876	1.527122	0.000547	159.893528	0.927081	
24877	2.396760	0.000521	283.647970	1.222822	

Visualize Customer Segmentation

In [44]:
```
# add clusters to the dataframe
customers["cluster_name"] = customers["clusters"].astype(str)
```

In [45]:
```
# visualize log_transformation customer segments with a 3D plot
fig = px.scatter_3d(customers,
                    x="log_products_ordered",
                    y="log_average_return_rate",
                    z="log_total_spending",
                    color='cluster_name',
                    hover_data=["products_ordered",
                                "average_return_rate",
                                "total_spending"],
```

```
                      category_orders = {"cluster_name":
                                         ["0", "1", "2", "3"]},
                      symbol = "is_center"
                      )

fig.update_layout(margin=dict(l=0, r=0, b=0, t=0))
fig.show()
```



Check for Cluster Magnitude

In [46]:
```
# values for log_transformation
cardinality_df = pd.DataFrame(
    customers.cluster_name.value_counts().reset_index())

cardinality_df.rename(columns={"index": "Customer Groups",
                               "cluster_name": "Customer Group Magnitude"},
                      inplace=True)
```

In [47]:
```
cardinality_df
```

Out[47]:

	Customer Groups	Customer Group Magnitude
0	1	10468
1	2	7206

	Customer Groups	Customer Group Magnitude
2	3	5116
3	0	2088

In [48]:

```
fig = px.bar(cardinality_df, x="Customer Groups",
             y="Customer Group Magnitude",
             color = "Customer Groups",
             category_orders = {"Customer Groups": ["0", "1", "2", "3"]})

fig.update_layout(xaxis = dict(
        tickmode = 'linear',
        tick0 = 1,
        dtick = 1),
                  yaxis = dict(
        tickmode = 'linear',
        tick0 = 1000,
        dtick = 1000))

fig.show()
```