ASSEGINMENT DATE	14 OCTBER 2022
STUDENT NAME	NAGINENI BHARGAV
STUDENT ROLL NUMBER	110719104032
MAXIMUM MARKS	2 MARKS

ASSIGNMENT :4

CUSTOMER SEGMENTATION ANALYSIS

There are approximately 25000 unique customers combined with their order information in the raw dataset:

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

Out[3]:

	product_title	product_type	variant_title	variant_sku	variant_id	customer_id	order_id	day	net_quantity	gross_sales	discounts	returns
0	DPR	DPR	100	AD-982-708-895-F-6C894FB	52039657	1312378	83290718932496	04/12/2018	2	200.0	-200.00	0.00
1	RJF	Product P	28 / A / MTM	83-490-E49-8C8-8-3B100BC	56914686	3715657	36253792848113	01/04/2019	2	190.0	-190.00	0.00
2	CLH	Product B	32 / B / FtO	68-ECA-BC7-3B2-A-E73DE1B	24064862	9533448	73094559597229	05/11/2018	0	164.8	-156.56	-8.24
3	NMA	Product F	40 / B / FtO	6C-1F1-226-1B3-2-3542B41	43823868	4121004	53616575668264	19/02/2019	1	119.0	-119.00	0.00
4	NMA	Product F	40 / B / FtO	6C-1F1-226-1B3-2-3542B41	43823868	4121004	29263220319421	19/02/2019	1	119.0	-119.00	0.00

Dataset is well-formatted and had no NA values. So, we can start by forming the features. 3 features will be calculated per `customer_id` and they will help us with the visualization (using Plotly library) and algorithm explainability in the latter steps. Data preparation will be done with pandas and numpy.

- **Number of products ordered:** It is calculated by counting the `product_type` ordered by a customer with the below function:

- **Average return rate:** It is the ratio of `returned_item_quantity` to the `ordered_item_quantity` averaged for all orders of a customer.

order_id	day	net_quantity	gross_sales	discounts	returns	net_sales	taxes	total_sales	returned_item_quantity	ordered_item_quantity
83290718932496	04/12/2018	2	200.0	-200.00	0.00	0.0	0.0	0.0	0	2
36253792848113	01/04/2019	2	190.0	-190.00	0.00	0.0	0.0	0.0	0	2
73094559597229	05/11/2018	0	164.8	-156.56	-8.24	0.0	0.0	0.0	-2	2
53616575668264	19/02/2019	1	119.0	-119.00	0.00	0.0	0.0	0.0	0	1
29263220319421	19/02/2019	1	119.0	-119.00	0.00	0.0	0.0	0.0	0	1

- **Total spending:** It is the aggregated sum of total sales, which is the final amount after taxes and returns.

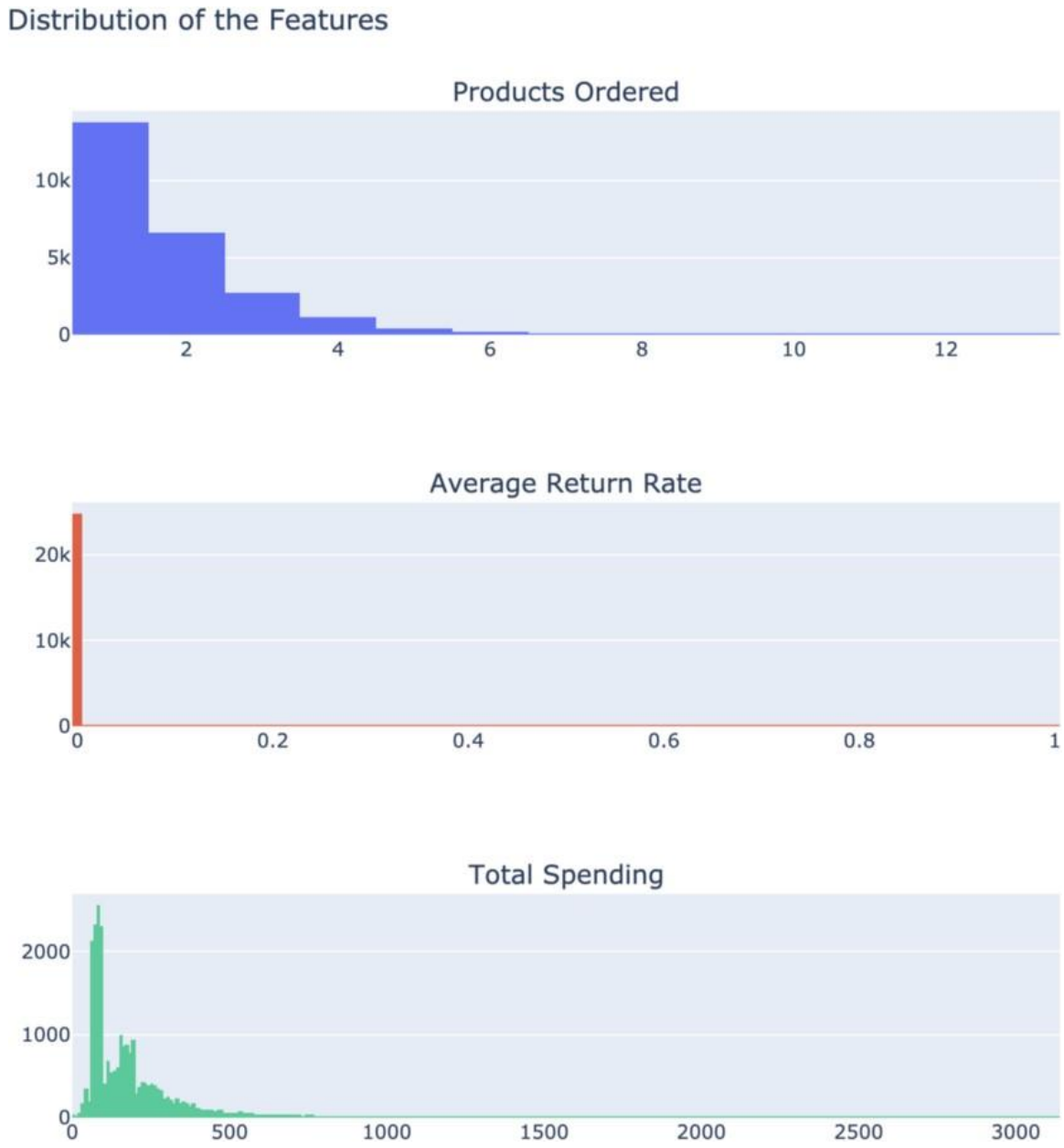
After the calculations, 3 features merged in the `customers` data frame:

```
In [20]: customers.head()
```

Out[20]:

	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

Let’s have a look at the individual distribution of the features:



All 3 distributions are positively skewed distributions. Products ordered shows a power-law distribution and average return rate of 99% of the customers are 0.

3 features have different ranges varying between [1, 13], [0, 1] and [0, 1000] which is an important observation showing that features need scaling!

Scaling:

K-means algorithm interprets each row in the customers data frame as a point in a 3-dimensional space. When grouping them, it uses the euclidian distance between the data points and the center of the group. With highly varying ranges, algorithm may perform poorly and not be able to form the groups as expected.

For K-means to perform effectively, we are going to scale the data using logarithmic transformation which is a suitable transformation for skewed data. This will scale down proportionally the 3D space which our data is spread, yet preserving the proximity between the points.

After applying the above function, customers **data frame is ready to be fed into K-means clustering:**

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

Out[28]:

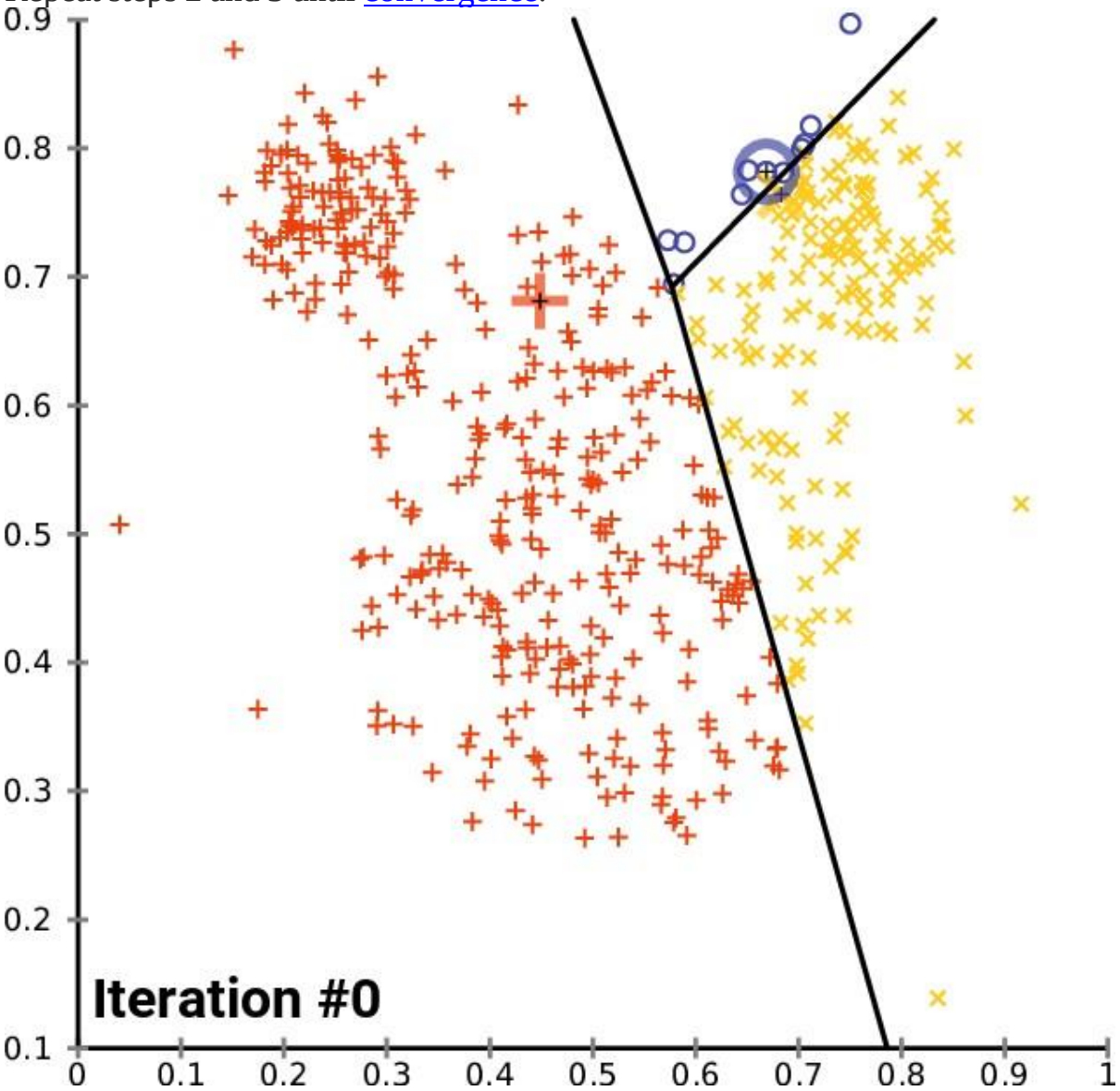
	products_ordered	average_return_rate	total_spending	log_products_ordered	log_average_return_rate	log_total_spending
0	1	0.0	260.0	0.693147	0.0	5.564520
1	1	0.0	79.2	0.693147	0.0	4.384524
2	3	0.0	234.2	1.386294	0.0	5.460436
3	1	0.0	89.0	0.693147	0.0	4.499810
4	2	0.0	103.0	1.098612	0.0	4.644391

3. Segmentation with K-means Clustering

We are going to use K-means algorithm from scikit-learn. Let's first understand how the algorithm will form customer groups:

1. Initialize *k=n centroids=number-of-clusters* randomly or smartly
2. Assign each data point to the closest centroid based on euclidian distance, thus forming the groups
3. Move centers to the average of all points in the cluster

Repeat steps 2 and 3 until convergence.



K-means in action with n centroids=3. Source: Wikimedia

While running the steps through, the algorithm checks the sum of squared distances between clustered-point and center for each cluster. Mathematically speaking, it tries to minimize — optimize the ***within-cluster sum-of-squared-distances*** or ***inertia*** of each cluster**.**

$$\sum_{i=0}^{n} \min_{\mu_j \in C}(||x_i - \mu_j||^2)$$

Mathematical expression of within-cluster sum-of-squared-distances or inertia where *X is the points in the cluster and μ is the current centroid*

When ***inertia*** value does not minimize further, algorithm converges. Thus, iteration stops.

```
from sklearn.cluster import Kmeans
kmeans_model = KMeans(init='k-means++',
                      max_iter=500,
                      random_state=42)
```

- init parameter with the k-means++ allows the algorithm to place initial centers smartly, rather than random.
- max_iter is the maximum number of iterations of the algorithm in a single run, default value is 300.
- random_state guarantees the reproducibility of the model results.

This algorithm is easy to understand, fits well to large datasets in terms of computing times and guarantees convergence. However, when centroids are initialized randomly, algorithm may not assign the points to the groups in the most optimal way.

One important consideration is the selection of *k*. In other words, how many groups should be formed? For example, K-means applied above uses *k=8* as a default value.

In the next step, we are going to choose k which is the most important hyperparameter of K-means.