CAPSTONE 3

Un-Supervised Learning

¶

Research Question

HELP TO SEGMENT THE PAYMENT TRANSACTION FOR AN ECOMMERCE WEBSITE

Content

- 1. THE DATA
- 1. CLEAN DATA
- 1. FEATURE ENGINNER
- 1. APPLY DIMENSIONALITY REDUCTION TECHNIQUES TO VISUALIZE THE OBSERVATIONS
- 1. APPLY CLUSTERING TECHNIQUES TO GROUP TOGETHER SIMILAR OBSERVATIONS

1- The Data

https://www.kaggle.com/olistbr/brazilian-ecommerce/data?select=olist_customers_dataset.csv (https://www.kaggle.com/olistbr/brazilian-ecommerce/data?select=olist_customers_dataset.csv)

In [1]:
```
import numpy as np
import pandas as pd
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
import umap
from sklearn import datasets, metrics
# Display preferences.
%matplotlib inline
pd.options.display.float_format = '{:.3f}'.format
```

In [2]:
```
items_df = pd.read_csv("order_items.csv")
payments_df = pd.read_csv("order_payments.csv")
orders_df = pd.read_csv("orders_dataset.csv")
customer_df = pd.read_csv("customers_dataset.csv")
```

In [3]:
```
df_pay = pd.merge(items_df, payments_df, on='order_id')
df_zipcode = pd.merge(orders_df, customer_df, on='customer_id')
```

In [4]:
```
df = pd.merge(df_zipcode, df_pay, on='order_id')
```

In [5]:
```
#display all columns in dataframe
from IPython.display import display
pd.options.display.max_columns= None
display(df)
```

	order_id	customer_id	order_status	order_purchase_timestamp	order_approve
0	e481f51cbdc54678b7cc49136f2d6af7	9ef432eb6251297304e76186b10a928d	delivered	2017-10-02 10:56:33	2017-1 11:0
1	e481f51cbdc54678b7cc49136f2d6af7	9ef432eb6251297304e76186b10a928d	delivered	2017-10-02 10:56:33	2017-1 11:0
2	e481f51cbdc54678b7cc49136f2d6af7	9ef432eb6251297304e76186b10a928d	delivered	2017-10-02 10:56:33	2017-1 11:0
3	53cdb2fc8bc7dce0b6741e2150273451	b0830fb4747a6c6d20dea0b8c802d7ef	delivered	2018-07-24 20:41:37	2018-0 03:2
4	47770eb9100c2d0c44946d9cf07ec65d	41ce2a54c0b03bf3443c3d931a367089	delivered	2018-08-08 08:38:49	2018-0 08:5
...	...	...	...	...	
117596	63943bddc261676b46f01ca7ac2f7bd8	1fca14ff2861355f6e5f14306ff977a7	delivered	2018-02-06 12:58:58	2018-0 13:1
117597	83c1379a015df1e13d02aae0204711ab	1aa71eb042121263aafbe80c1b562c9c	delivered	2017-08-27 14:46:43	2017-0 15:0
117598	11c177c8e97725db2631073c19f07b62	b331b74b18dc79bcdf6532d51e1637c1	delivered	2018-01-08 21:28:27	2018-0 21:3
117599	11c177c8e97725db2631073c19f07b62	b331b74b18dc79bcdf6532d51e1637c1	delivered	2018-01-08 21:28:27	2018-0 21:3
117600	66dea50a8b16d9b4dee7af250b4be1a5	edb027a75a1449115f6b43211ae02a24	delivered	2018-03-08 20:57:30	2018-0 11:2

117601 rows × 22 columns

In [6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 117601 entries, 0 to 117600
Data columns (total 22 columns):
order_id                        117601 non-null object
customer_id                     117601 non-null object
order_status                    117601 non-null object
order_purchase_timestamp        117601 non-null object
order_approved_at               117586 non-null object
order_delivered_carrier_date    116356 non-null object
order_delivered_customer_date   115034 non-null object
order_estimated_delivery_date   117601 non-null object
customer_unique_id              117601 non-null object
customer_zip_code_prefix        117601 non-null int64
customer_city                   117601 non-null object
customer_state                  117601 non-null object
order_item_id                   117601 non-null int64
product_id                      117601 non-null object
seller_id                       117601 non-null object
shipping_limit_date             117601 non-null object
price                           117601 non-null float64
freight_value                   117601 non-null float64
payment_sequential              117601 non-null int64
payment_type                    117601 non-null object
payment_installments            117601 non-null int64
payment_value                   117601 non-null float64
dtypes: float64(3), int64(4), object(15)
memory usage: 20.6+ MB
```

2. CLEAN DATA

In [7]:
```
total_missing= df.isnull().sum().sort_values(ascending= False)
percent_missing = (df.isnull().sum()/df.isnull().count()).sort_values(ascending= False)
missing_data = pd.concat ([total_missing, percent_missing], axis=1, keys = ['Total', 'Percent'])
missing_data.head(20)
```

Out[7]:

	Total	Percent
order_delivered_customer_date	2567	0.022
order_delivered_carrier_date	1245	0.011
order_approved_at	15	0.000
payment_value	0	0.000
payment_installments	0	0.000
customer_id	0	0.000
order_status	0	0.000
order_purchase_timestamp	0	0.000
order_estimated_delivery_date	0	0.000
customer_unique_id	0	0.000
customer_zip_code_prefix	0	0.000
customer_city	0	0.000
customer_state	0	0.000
order_item_id	0	0.000
product_id	0	0.000
seller_id	0	0.000
shipping_limit_date	0	0.000
price	0	0.000
freight_value	0	0.000
payment_sequential	0	0.000

Take out the letters for order_id and product_id , shorten the numbers, convert string into int, and put it into a new column

In [8]:
```
#took off the letters/string types
df['order_id'] = df['order_id'].str.replace(r'\D', '')
df['order_short_id'] = df['order_id'].str[-5:]
df['order_short_id'].astype(int)
df
```

Out[8]:

	order_id	customer_id	order_status	order_purchase_timestamp	order_approved_at	orc
0	4815154678749136267	9ef432eb6251297304e76186b10a928d	delivered	2017-10-02 10:56:33	2017-10-02 11:07:15	
1	4815154678749136267	9ef432eb6251297304e76186b10a928d	delivered	2017-10-02 10:56:33	2017-10-02 11:07:15	
2	4815154678749136267	9ef432eb6251297304e76186b10a928d	delivered	2017-10-02 10:56:33	2017-10-02 11:07:15	
3	53287067412150273451	b0830fb4747a6c6d20dea0b8c802d7ef	delivered	2018-07-24 20:41:37	2018-07-26 03:24:27	
4	477709100204494690765	41ce2a54c0b03bf3443c3d931a367089	delivered	2018-08-08 08:38:49	2018-08-08 08:55:23	
...	...	...	...	...	...	
117596	6394326167646017278	1fca14ff2861355f6e5f14306ff977a7	delivered	2018-02-06 12:58:58	2018-02-06 13:10:37	
117597	831379015113020204711	1aa71eb042121263aafbe80c1b562c9c	delivered	2017-08-27 14:46:43	2017-08-27 15:04:16	
117598	111778977252631073190762	b331b74b18dc79bcdf6532d51e1637c1	delivered	2018-01-08 21:28:27	2018-01-08 21:36:21	
117599	111778977252631073190762	b331b74b18dc79bcdf6532d51e1637c1	delivered	2018-01-08 21:28:27	2018-01-08 21:36:21	
117600	6650816947250415	edb027a75a1449115f6b43211ae02a24	delivered	2018-03-08 20:57:30	2018-03-09 11:20:28	

117601 rows × 23 columns

Check that features do not have any unique values assigned to them

In [9]:
```
print('Unique values for customer_zip_code_prefix :', df['customer_zip_code_prefix'].unique())
print('Unique values for order_item_id :', df['order_item_id'].unique())
print('Unique values for price :', df['price'].unique())
print('Unique values for freight_value :', df['freight_value'].unique())
print('Unique values for payment_type :', df['payment_type'].unique())
print('Unique values for payment_installments :', df['payment_installments'].unique())
print('Unique values for payment_value :', df['payment_value'].unique())
```

```
Unique values for customer_zip_code_prefix : [ 3149 47813 75265 ... 83870  5127 45920]
Unique values for order_item_id : [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21]
Unique values for price : [ 29.99 118.7  159.9  ...  91.09 109.66 213.39]
Unique values for freight_value : [  8.72  22.76  19.22 ... 173.23   5.01   0.8 ]
Unique values for payment_type : ['credit_card' 'voucher' 'boleto' 'debit_card']
Unique values for payment_installments : [ 1  3  6 10  4  2  8  9  7  5 13 12 15 14 21 18 24 17 11 2
0 23 16 22  0]
Unique values for payment_value : [ 18.12   2.    18.59 ... 173.63 271.01 441.16]
```

3. Feature engineer

About the features

- price = item price
- payment_installments = number of installments chosen by the customer
- payment_value = is full payment including freight for each item
- freight_value = item freight value item (if an order has more than one item the freight value is splitted between items)

- Group by order id and customer zip code, and get the sum of price, payment installments, payment value, and freight value.
- Drop order id

In [10]:
```
df= df.groupby(['order_short_id','customer_zip_code_prefix'], as_index=False)['price','payment_instal
lments','payment_value','freight_value'].sum()
```

In [11]:
```
df.drop(columns= ['order_short_id'], inplace= True)
```

In [12]:
```
df
```

Out[12]:

	customer_zip_code_prefix	price	payment_installments	payment_value	freight_value
0	13044	257.000	1	269.420	12.420
1	87303	37.900	2	54.690	16.790
2	7032	198.000	2	213.140	15.140
3	4617	59.990	3	68.100	8.110
4	15600	31.900	4	44.590	12.690
...	...	...	...	...	...
98655	3410	154.000	4	341.200	16.600
98656	24220	34.990	1	60.370	25.380
98657	37757	39.900	1	55.000	15.100
98658	8110	105.000	1	121.700	16.700
98659	90040	120.000	3	164.610	44.610

98660 rows × 5 columns

4. Apply dimensionality reduction techniques to visualize the observations

4.1 UMAP

UMAP: Visualization

In [13]:
```
#standarize data
from sklearn.preprocessing import StandardScaler

sc= StandardScaler()

X_std = sc.fit_transform(df)
```

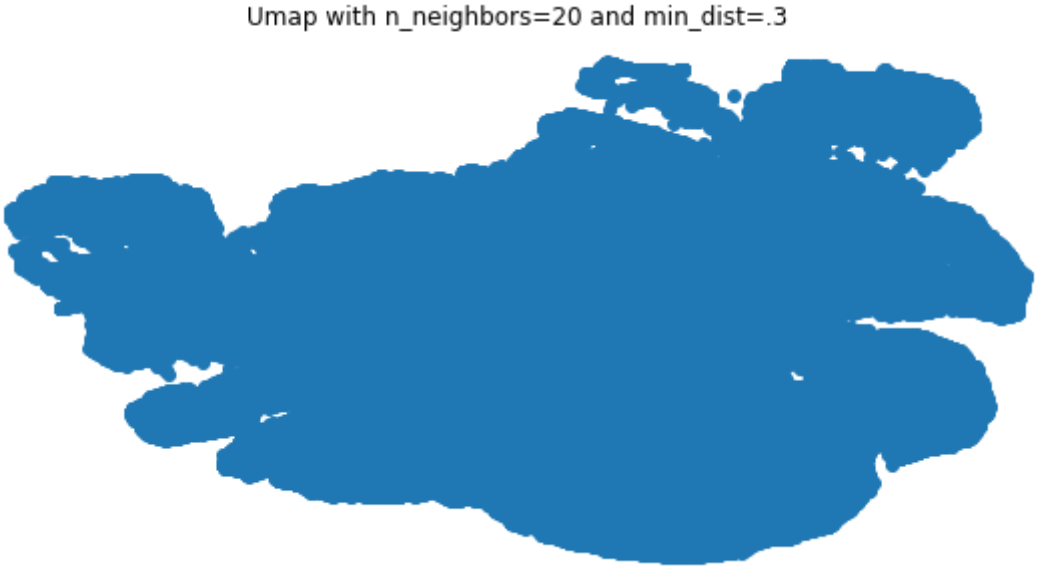
In [14]:
```
umap_results = umap.UMAP(n_neighbors=10,
                         min_dist=.3,
                         metric='correlation').fit_transform(X_std)

fig, ax = plt.subplots(figsize=(8,6))
plt.title("Umap standardized with n_neighbors=10 and min_dist=.3")
plt.scatter(umap_results[:, 0], umap_results[:, 1])
plt.xticks([])
plt.yticks([])
plt.axis('off')
plt.show()
```

```
/usr/local/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:355: NumbaWarning: The TBB threadi
ng layer requires TBB version 2019.5 or later i.e., TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERF
ACE_VERSION = 11000. The TBB threading layer is disabled.
  warnings.warn(problem)
```

In [15]:
```
umap_results = umap.UMAP(n_neighbors=20,
                         min_dist=.3,
                         metric='correlation').fit_transform(X_std)

plt.figure(figsize=(10,5))
plt.title("Umap with n_neighbors=20 and min_dist=.3")
plt.scatter(umap_results[:, 0], umap_results[:, 1])
plt.xticks([])
plt.yticks([])
plt.axis('off')
plt.show()
```



4.2 PCA

PCA: with 80% variance and 1st and 2nd components

In [16]:
```
#80% of variance
from sklearn.decomposition import PCA
pca = PCA(n_components = 0.80)
pca.fit(X_std)
reduced = pca.transform(X_std)
pca_components = pca.fit_transform(X_std)
```

In [17]:
```
#plot pca
plt.figure(figsize=(10,5))
plt.scatter(pca_components[:, 0], pca_components[:, 1])
plt.xticks([])
plt.yticks([])
plt.axis('off')
plt.show()
```

PCA: with 80% variance and 1st and 3rd components

In [18]:
```
#plot pca
plt.figure(figsize=(10,5))
plt.scatter(pca_components[:, 0], pca_components[:, 2])
plt.xticks([])
plt.yticks([])
plt.axis('off')
plt.show()
```

I tried applying different parameters to both PCA and UMAP but as shown, nothing really captures a clear view of the clusters for the data. In this case UMAP worked better than PCA.

5. Apply clustering techniques to group together similar observations

5.1 Kmeans clustering

Try different types of kmeans hyperparameters with the number of clusters.

In [19]:
```
kmeans_2 = KMeans(n_clusters=2, random_state=300).fit_predict(X_std)
kmeans_6 = KMeans(n_clusters=6, random_state=300).fit_predict(X_std)
kmeans_10 = KMeans(n_clusters=10, random_state=300).fit_predict(X_std)
```

In [20]:
```
print("Silhouette score for two cluster k-means: {}".format(
    metrics.silhouette_score(X_std, kmeans_2, metric='euclidean')))
print("Silhouette score for six cluster k-means: {}".format(
    metrics.silhouette_score(X_std, kmeans_6, metric='euclidean')))
print("Silhouette score for ten cluster k-means: {}".format(
    metrics.silhouette_score(X_std, kmeans_10, metric='euclidean')))
```

```
Silhouette score for two cluster k-means: 0.6840142305407634
Silhouette score for six cluster k-means: 0.4671462595715169
Silhouette score for ten cluster k-means: 0.4390797013175752
```

Kmeans with two clusters score the highest and therefor perform the best. It seems the higher the number of cluster the lower the scores goes.

Next, visualize kmeans with a smaller set of the data.

Since n_cluster= 2 is the highest we will visualize the clusters

In [21]:
```
# Fitting K-Means to the dataset
kmeans = KMeans(n_clusters = 2, init = 'k-means++', random_state = 40)
# Compute cluster centers and predict cluster index for each sample.
y_km = kmeans.fit_predict(X_std)
```

In [22]:
```
#plot only a few rows for better understanding and visualization of the graph
X_t = X_std[0:1000,0:1000]
y_t = y_km[0:1000]
```

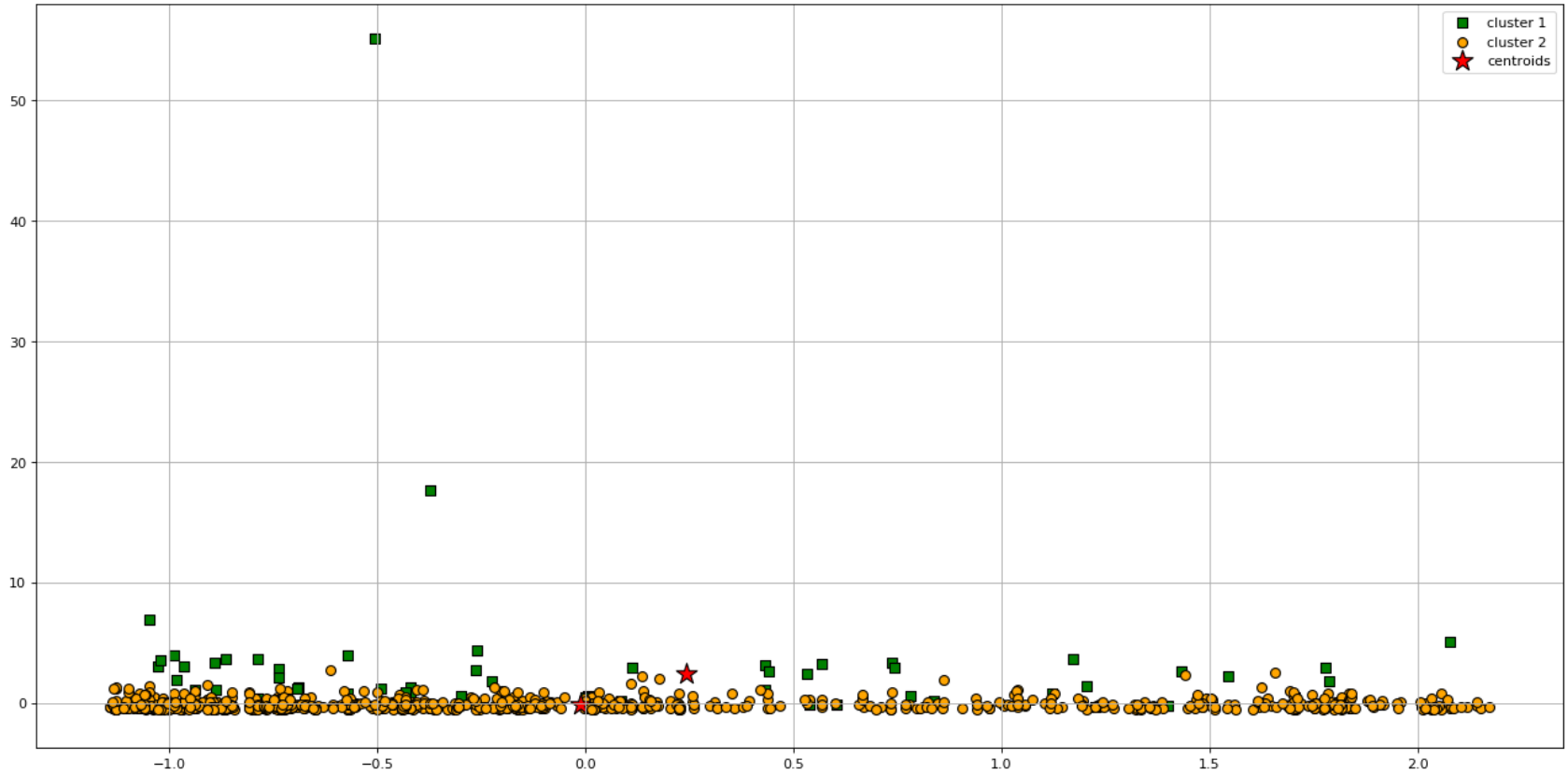
In [23]:
```
plt.figure(figsize=(20, 10), dpi=80)

plt.scatter(
    X_t[y_t == 0, 0], X_t[y_t == 0, 1],
    s=50, c='green',
    marker='s', edgecolor='black',
    label='cluster 1'
)

plt.scatter(
    X_t[y_t == 1, 0], X_t[y_t == 1, 1],
    s=50, c='orange',
    marker='o', edgecolor='black',
    label='cluster 2'
)



# plot the centroids
plt.scatter(
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
    s=250, marker='*',
    c='red', edgecolor='black',
    label='centroids'
)
plt.legend(scatterpoints=1)
plt.grid()
plt.show()
```



5.2 GMM clustering

In [24]:
```
from sklearn.mixture import GaussianMixture
# Defining the agglomerative clustering
gmm_cluster = GaussianMixture(n_components=2, random_state=123)

# Fit model
clusters = gmm_cluster.fit_predict(X_std)
```

In [25]:
```
print("The silhoutte score of the GMM solution: {}"
      .format(metrics.silhouette_score(X_std, clusters, metric='euclidean')))
```

```
The silhoutte score of the GMM solution: 0.40474589429030117
```

In [26]:
```
# gmm with different parameters
gmm_full = GaussianMixture(n_components=2, random_state=123, covariance_type="full").fit_predict(X_st
d)
gmm_tied = GaussianMixture(n_components=2, random_state=123, covariance_type="tied").fit_predict(X_st
d)
gmm_diag = GaussianMixture(n_components=2, random_state=123, covariance_type="diag").fit_predict(X_st
d)
gmm_sph = GaussianMixture(n_components=2, random_state=123, covariance_type="spherical").fit_predict(
X_std)
```

In [27]:
```
print("The silhoutte score of the GMM full: {}"
      .format(metrics.silhouette_score(X_std, gmm_full, metric='euclidean')))
print("The silhoutte score of the GMM tied: {}"
      .format(metrics.silhouette_score(X_std, gmm_tied, metric='euclidean')))
print("The silhoutte score of the GMM diag: {}"
      .format(metrics.silhouette_score(X_std, gmm_diag, metric='euclidean')))
print("The silhoutte score of the GMM spherical: {}"
      .format(metrics.silhouette_score(X_std, gmm_sph, metric='euclidean')))
```

```
The silhoutte score of the GMM full: 0.40474589429030117
The silhoutte score of the GMM tied: 0.7608676685807885
The silhoutte score of the GMM diag: 0.4379828846411668
The silhoutte score of the GMM spherical: 0.6289668149154796
```

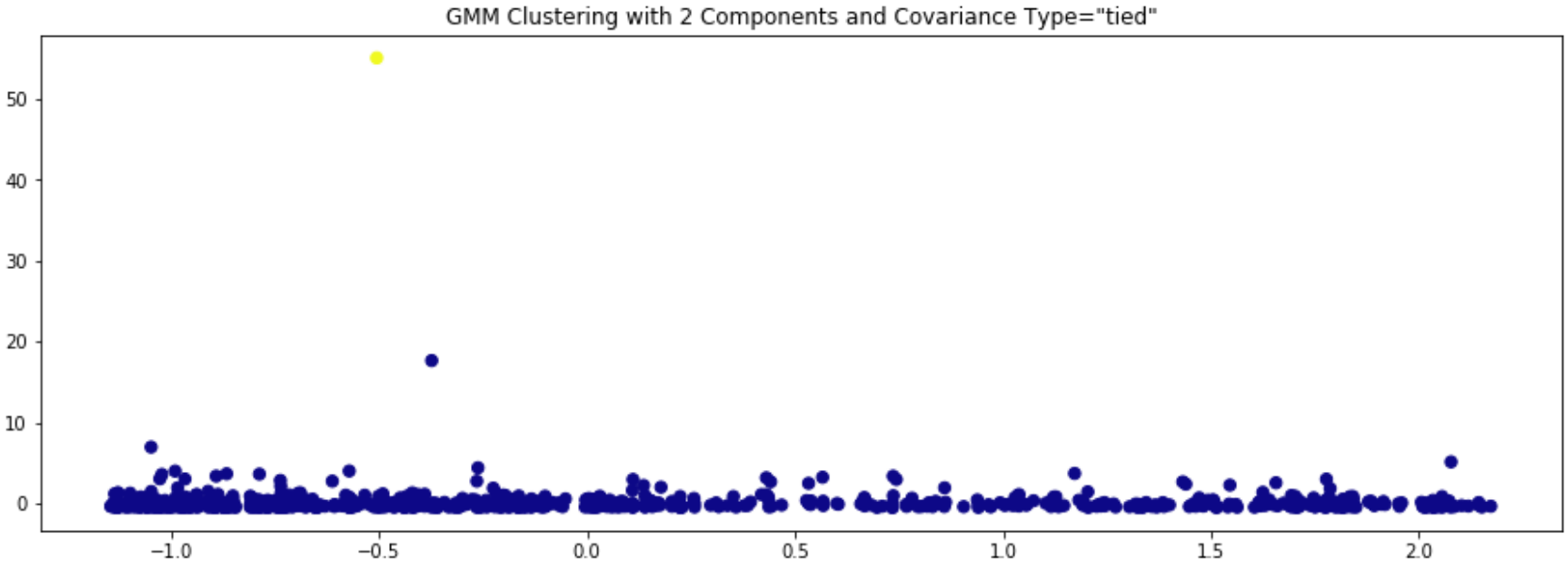
Covariance type of tied and 2 clusters, is the one with the highest score at 76%. Also GMM score higher than Kmeans.

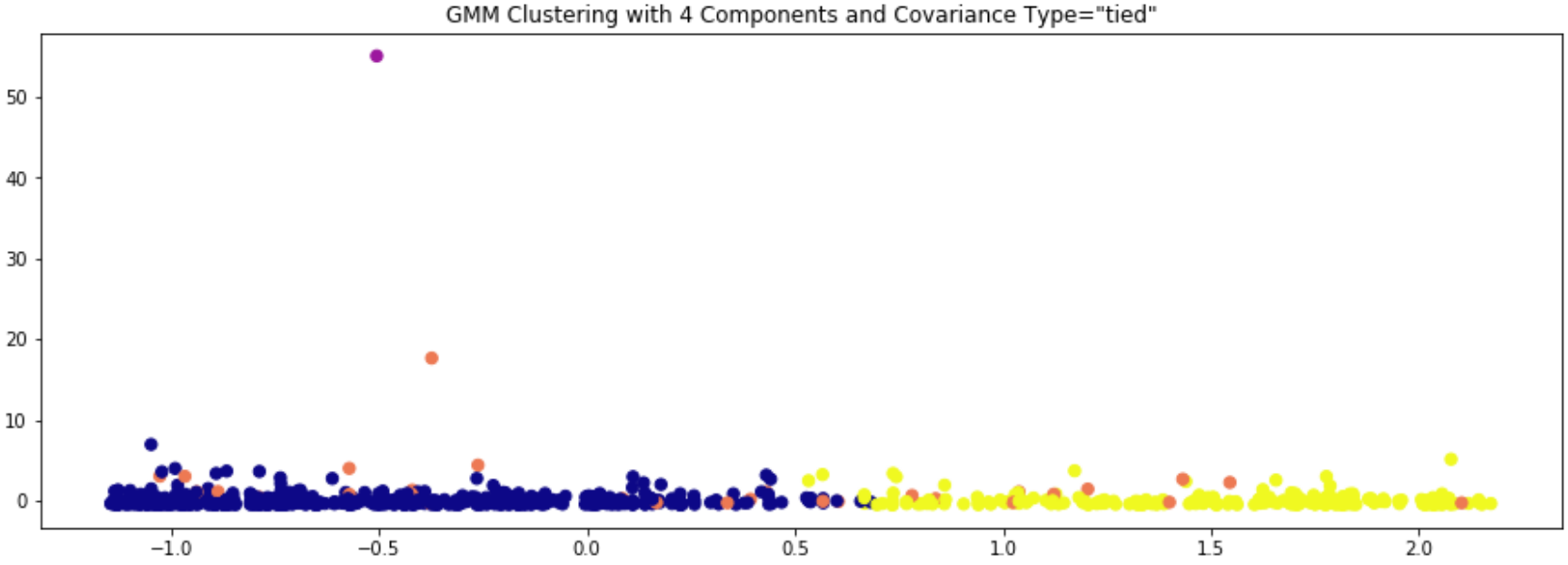
For a better visualization of GMM, cut down the amount of data to 1000

In [28]:
```
X_t = X_std[0:1000,0:1000]
y_t = y_km[0:1000]
```

In [29]:
```
fig, ax = plt.subplots(figsize=(15,5))
gmm = GaussianMixture(n_components=2,random_state=123, covariance_type="tied").fit(X_t)
labels = gmm.predict(X_t)
ax.set_title('GMM Clustering with 2 Components and Covariance Type="tied"')
plt.scatter(X_t[:, 0], X_t[:, 1], c=labels, s=40, cmap='plasma');

fig, ax = plt.subplots(figsize=(15,5))
gmm = GaussianMixture(n_components=4,random_state=123, covariance_type="tied").fit(X_t)
ax.set_title('GMM Clustering with 4 Components and Covariance Type="tied"')
labels = gmm.predict(X_t)
plt.scatter(X_t[:, 0], X_t[:, 1], c=labels, s=40, cmap='plasma');
```


GMM Clustering with 2 Components and Covariance Type="tied"
50
40
30
20
10
0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


GMM Clustering with 4 Components and Covariance Type="tied"
50
40
30
20
10
0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

Conclusion

To visualize the clustering with this data has been quiet challenging. Nothing seemed to work very well, and because of time I couldn't go back and find another data. I decided to just stick with this one and see what happens. I figure not every data that I will received will be perfect anyways.

When visualization the data with UMAP or PCA, both did not do much of a good job. I tried playing around with the parameters for UMAP, which it took a long time to run each test. I finally decided to stick to only two UMAP. I think overall UMAP did a better job than PCA in visualization of the clusters.

Last, I decided to test the data with kmeans and GMM. Between the two GMM score better at 76%, almost 10 points above kmeans. When it came down to the graph, I decided to break down the data to smaller set. In the case of the graphs kmeans showed better the clustering for 2 than GMM.

Use of data: Clustering understand ecommerce pattern. With more time, the graphs can display what transaction trends are happening in different zip code.

Next steps: Apply other clustering techniques like hierarchical clustering and show more graphs. Break down the features into maybe only two, to see a better clustering vision. For example, see the clusters for freight and payment value.

Research question: Figure out the clustering that have the highest transaction. Also, break down the features into maybe only two, to see a better clustering vision. For example, see the clusters for freight and payment value.