ADS505 -- Assignment 5.1

Segmenting Consumers of Bath Soap

Stephen Kuc

10/10/22

In [89]:
```
# importing necessary libraries

import numpy as np
import pandas as pd

from sklearn import preprocessing
from sklearn.preprocessing import Normalizer, StandardScaler
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.metrics import pairwise, accuracy_score
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from sklearn.cluster import KMeans
from pandas.plotting import parallel_coordinates
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

import dmba
from dmba import classificationSummary, plotDecisionTree

import matplotlib.pylab as plt
import seaborn as sns
```

In [18]:
```
# Import required packages for this chapter
import pandas as pd
import numpy as np

from scipy.spatial.distance import cosine
from sklearn.metrics.pairwise import cosine_similarity

from surprise import Dataset
from surprise import Reader
from surprise import KNNBasic
```

In [19]:
```
# Filtering warnings
import warnings
warnings.filterwarnings("ignore")
```

Overall problem -- CRISA wants to segment the market based on (1) purchase behavior and (2) basis of purchase; and then use this to deploy promotional budgets more effectively, among other benefits and uses. Measuring brand loyalty will be an important part of this as well.

Before diving in, let's load the data and do some basic EDA

In [20]:
```
#loading data

df = pd.read_csv("C:/Users/steph/OneDrive/Documents/USD/ADS505/Data/BathSoapHousehold.csv")
df.head()
```

Out[20]:

	Member id	SEC	FEH	MT	SEX	AGE	EDU	HS	CHILD	CS	...	PropCat 6	PropCat 7	PropCat 8	PropCat 9	PropCat 10	PropCat 11	PropCat 12	PropCa 1
0	1010010	4	3	10	1	4	4	2	4	1	...	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.028037	0.(
1	1010020	3	2	10	2	2	4	4	2	1	...	0.347048	0.026834	0.016100	0.014311	0.0	0.059034	0.000000	0.(
2	1014020	2	3	10	2	4	5	6	4	1	...	0.121212	0.033550	0.010823	0.008658	0.0	0.000000	0.016234	0.(
3	1014030	4	0	0	0	4	0	0	5	0	...	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.(
4	1014190	4	1	10	2	3	4	4	3	1	...	0.000000	0.000000	0.048193	0.000000	0.0	0.000000	0.000000	0.(

5 rows × 46 columns

In [21]:
```
print(df.dtypes)
```

```
Member id                 int64
SEC                       int64
FEH                       int64
MT                        int64
SEX                       int64
AGE                       int64
EDU                       int64
HS                        int64
CHILD                     int64
CS                        int64
Affluence Index           int64
No. of Brands             int64
Brand Runs                int64
Total Volume              int64
No. of  Trans             int64
Value                   float64
Trans / Brand Runs      float64
Vol/Tran                float64
Avg. Price              float64
Pur Vol No Promo - %    float64
Pur Vol Promo 6 %       float64
Pur Vol Other Promo %   float64
Br. Cd. 57, 144         float64
Br. Cd. 55              float64
Br. Cd. 272             float64
Br. Cd. 286             float64
Br. Cd. 24              float64
Br. Cd. 481             float64
Br. Cd. 352             float64
Br. Cd. 5               float64
Others 999              float64
Pr Cat 1                float64
Pr Cat 2                float64
Pr Cat 3                float64
Pr Cat 4                float64
PropCat 5               float64
PropCat 6               float64
PropCat 7               float64
PropCat 8               float64
PropCat 9               float64
PropCat 10              float64
PropCat 11              float64
PropCat 12              float64
PropCat 13              float64
PropCat 14              float64
PropCat 15              float64
dtype: object
```

In [22]:
```
df.isnull().sum()
```

Out[22]:
```
Member id                0
SEC                      0
FEH                      0
MT                       0
SEX                      0
AGE                      0
EDU                      0
HS                       0
CHILD                    0
CS                       0
Affluence Index          0
No. of Brands            0
Brand Runs               0
Total Volume             0
No. of  Trans            0
Value                    0
Trans / Brand Runs       0
Vol/Tran                 0
Avg. Price               0
Pur Vol No Promo - %     0
Pur Vol Promo 6 %        0
Pur Vol Other Promo %    0
Br. Cd. 57, 144          0
Br. Cd. 55               0
Br. Cd. 272              0
Br. Cd. 286              0
Br. Cd. 24               0
Br. Cd. 481              0
Br. Cd. 352              0
Br. Cd. 5                0
Others 999               0
Pr Cat 1                 0
Pr Cat 2                 0
Pr Cat 3                 0
Pr Cat 4                 0
PropCat 5                0
PropCat 6                0
PropCat 7                0
PropCat 8                0
PropCat 9                0
PropCat 10               0
PropCat 11               0
PropCat 12               0
PropCat 13               0
PropCat 14               0
PropCat 15               0
dtype: int64
```

Since we'll be using K-means on different sets of variables, let's examine each set, and then recombine them when the time comes for that.

First, a set of variables that describe purchase behavior, including brand loyalty.

Purchase behavior also inclused - volume, frequency, susceptibility to discounts, and brand loyalty.

Viewing the features:

No. of Brands, Brand Runs, Total volume, Sum of Value, Trans/Brand Runs, Vol/Trans, Pur Vol, No Promo %, Pur Vol Promo 6%, Pur Vol Other Promo %, and all brandwise purchase (which may be further analyzed on its on)

In [23]:
```
# demographics dataframe, that may not be used
demographics = ['SEC', 'FEH' , 'MT', 'SEX', 'AGE', 'EDU', 'HS', 'SHILD', 'CS', 'Affluence_Index']

# purchase behavior dataframe, initially without brand
pb_df = df[['No. of Brands','Brand Runs', 'Total Volume',
            'Value', 'Trans / Brand Runs', 'Vol/Tran',
            'Pur Vol No Promo - %', 'Pur Vol Promo 6 %',
            'Pur Vol Other Promo %']]

# brand dataframe - will likely transform

brand_pct = df[['Br. Cd. 57, 144','Br. Cd. 55', 'Br. Cd. 272', 'Br. Cd. 286',
               'Br. Cd. 24', 'Br. Cd. 481', 'Br. Cd. 352', 'Br. Cd. 5', 'Others 999']]

brand_pct.head()
```

Out[23]:

	Br. Cd. 57, 144	Br. Cd. 55	Br. Cd. 272	Br. Cd. 286	Br. Cd. 24	Br. Cd. 481	Br. Cd. 352	Br. Cd. 5	Others 999
0	0.376947	0.130841	0.0	0.000000	0.0	0.000000	0.0	0.000000	0.492212
1	0.021467	0.075134	0.0	0.000000	0.0	0.059034	0.0	0.144902	0.699463
2	0.025974	0.545455	0.0	0.030303	0.0	0.000000	0.0	0.019481	0.378788
3	0.400000	0.600000	0.0	0.000000	0.0	0.000000	0.0	0.000000	0.000000
4	0.048193	0.144578	0.0	0.000000	0.0	0.000000	0.0	0.000000	0.807229

Since we are just interested in measuring brand loyalty, as mentioned in a note in question, a good way to see if one customer does really prefer one brand over another, we can just check for the max in the brand_pct dataframe, without using the Others 999 column, as that encompasses all other brands, not just one singular.

Before that, let's at least try to find the distances between brands

In [24]:
```
# before we simply use this, let's see if a distance metric can be found from the brand percentages

brand_pct_sp = brand_pct.drop('Others 999', axis = 1)

brand_pct_sp = brand_pct_sp.apply(lambda x: x.astype(float))

brand_distance = pairwise.pairwise_distances(brand_pct_sp, metric = 'euclidean')

brand_distance = pd.DataFrame(brand_distance)

brand_distance
```

Out[24]:

	0	1	2	3	4	5	6	7	8	9	...	590	591	592	5
0	0.000000	0.392366	0.544412	0.469725	0.329041	0.306842	0.114276	0.739209	0.131092	0.508000	...	0.163751	0.359799	1.043335	0.4408
1	0.392366	0.000000	0.491279	0.665772	0.173258	0.166288	0.459365	0.732421	0.402890	0.271948	...	0.485995	0.191845	0.978114	0.2564
2	0.544412	0.491279	0.000000	0.379695	0.403104	0.483485	0.658258	0.248089	0.654030	0.574144	...	0.707680	0.556869	1.108527	0.5784
3	0.469725	0.665772	0.379695	0.000000	0.575480	0.624051	0.557281	0.404554	0.600186	0.708862	...	0.604720	0.698034	1.203885	0.7450
4	0.329041	0.173258	0.403104	0.575480	0.000000	0.083679	0.416172	0.645765	0.366574	0.403563	...	0.451018	0.190259	0.975994	0.2416
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
595	0.966063	0.854518	1.051035	1.136673	0.909086	0.900526	0.986933	1.195881	0.960008	0.991236	...	0.995669	0.815742	1.299179	0.9179
596	0.432973	0.169459	0.566833	0.740444	0.226897	0.196341	0.483277	0.809146	0.420144	0.354396	...	0.504253	0.201571	0.886487	0.1731
597	0.140343	0.294062	0.428313	0.447030	0.202174	0.205099	0.246272	0.643167	0.227962	0.444924	...	0.291819	0.284207	0.996553	0.3506
598	0.378558	0.163827	0.439465	0.633826	0.065770	0.092369	0.457624	0.683133	0.401134	0.409041	...	0.488528	0.178553	0.970558	0.2186
599	0.608482	0.639730	0.800657	0.848629	0.623931	0.602296	0.618200	0.997481	0.596226	0.733603	...	0.625002	0.526648	1.148098	0.6511

600 rows × 600 columns

This above is comparing how similar each customer is to another in terms of how much they like each brand. While this could help us with clustering, to derive a variable from it won't be meaningful for brand loyalty. We will use the maximum brand percentage as the representation for that dataframe

In [25]:
```
brand_pct_sp = brand_pct.drop('Others 999', axis = 1)
brand_pct['max_brand_pct'] = brand_pct_sp.max(axis = 1)

brand_pct = brand_pct.drop(brand_pct_sp, axis = 1)

brand_pct
```

Out[25]:

	Others 999	max_brand_pct
0	0.492212	0.376947
1	0.699463	0.144902
2	0.378788	0.545455
3	0.000000	0.600000
4	0.807229	0.144578
...	...	...
595	0.058172	0.897507
596	0.678218	0.103960
597	0.557560	0.246564
598	0.863636	0.109091
599	0.186992	0.577236

600 rows × 2 columns

In [26]:
```
pb_df.head()
```

Out[26]:

	No. of Brands	Brand Runs	Total Volume	Value	Trans / Brand Runs	Vol/Tran	Pur Vol No Promo - %	Pur Vol Promo 6 %	Pur Vol Other Promo %
0	3	17	8025	818.0	1.41	334.38	1.000000	0.000000	0.000000
1	5	25	13975	1681.5	1.60	349.38	0.887299	0.096601	0.016100
2	5	37	23100	1950.0	1.70	366.67	0.941558	0.019481	0.038961
3	2	4	1500	114.0	1.00	375.00	1.000000	0.000000	0.000000
4	3	6	8300	591.0	2.17	638.46	0.614458	0.144578	0.240964

In [27]:
```
# now let's concat brand_pct with the rest of the purchase behaviors

purch_behavior = pd.concat([pb_df, brand_pct], axis = 1)

purch_behavior.head()
```

Out[27]:

	No. of Brands	Brand Runs	Total Volume	Value	Trans / Brand Runs	Vol/Tran	Pur Vol No Promo - %	Pur Vol Promo 6 %	Pur Vol Other Promo %	Others 999	max_brand_pct
0	3	17	8025	818.0	1.41	334.38	1.000000	0.000000	0.000000	0.492212	0.376947
1	5	25	13975	1681.5	1.60	349.38	0.887299	0.096601	0.016100	0.699463	0.144902
2	5	37	23100	1950.0	1.70	366.67	0.941558	0.019481	0.038961	0.378788	0.545455
3	2	4	1500	114.0	1.00	375.00	1.000000	0.000000	0.000000	0.000000	0.600000
4	3	6	8300	591.0	2.17	638.46	0.614458	0.144578	0.240964	0.807229	0.144578

Now that purch_behavior is subsetted, let's create the basis for purchase dataframe.

The features that will be included here will be all of the Price category wise purchase and Selling proposition wise purchase, as well as Avg. Price

In [28]:
```
df.columns
```

Out[28]:
```
Index(['Member id', 'SEC', 'FEH', 'MT', 'SEX', 'AGE', 'EDU', 'HS', 'CHILD',
       'CS', 'Affluence Index', 'No. of Brands', 'Brand Runs', 'Total Volume',
       'No. of  Trans', 'Value', 'Trans / Brand Runs', 'Vol/Tran',
       'Avg. Price ', 'Pur Vol No Promo - %', 'Pur Vol Promo 6 %',
       'Pur Vol Other Promo %', 'Br. Cd. 57, 144', 'Br. Cd. 55', 'Br. Cd. 272',
       'Br. Cd. 286', 'Br. Cd. 24', 'Br. Cd. 481', 'Br. Cd. 352', 'Br. Cd. 5',
       'Others 999', 'Pr Cat 1', 'Pr Cat 2', 'Pr Cat 3', 'Pr Cat 4',
       'PropCat 5', 'PropCat 6', 'PropCat 7', 'PropCat 8', 'PropCat 9',
       'PropCat 10', 'PropCat 11', 'PropCat 12', 'PropCat 13', 'PropCat 14',
       'PropCat 15'],
      dtype='object')
```

In [29]:
```
b_of_p_vars = df.columns[-15:].tolist()
b_of_p_vars.append('Avg. Price ')
```

In [30]:
```
# Basis of purchase utilizing variables features mentioned earlier

b_of_p = df[b_of_p_vars]

b_of_p.head()
```

Out[30]:

	Pr Cat 1	Pr Cat 2	Pr Cat 3	Pr Cat 4	PropCat 5	PropCat 6	PropCat 7	PropCat 8	PropCat 9	PropCat 10	PropCat 11	PropCat 12	PropCat 13	PropCat 14	Pr
0	0.233645	0.560748	0.130841	0.074766	0.501558	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.028037	0.0	0.130841	0.3
1	0.293381	0.547406	0.094812	0.064401	0.456172	0.347048	0.026834	0.016100	0.014311	0.0	0.059034	0.000000	0.0	0.080501	0.0
2	0.120130	0.318182	0.561688	0.000000	0.244589	0.121212	0.033550	0.010823	0.008658	0.0	0.000000	0.016234	0.0	0.561688	0.0
3	0.000000	0.400000	0.600000	0.000000	0.400000	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.600000	0.0
4	0.000000	0.048193	0.144578	0.807229	0.807229	0.000000	0.000000	0.048193	0.000000	0.0	0.000000	0.000000	0.0	0.144578	0.0

Now that each DF is created, let's scale each so we can use K-means clustering

In [31]:
```
scaler_bp = StandardScaler()
scaler_pb = StandardScaler()

scaler_bp.fit(b_of_p)
scaler_pb.fit(purch_behavior)

b_of_p_scaled = scaler_bp.transform(b_of_p)
purch_behavior_scaled = scaler_pb.transform(purch_behavior)

b_of_p_scaled = pd.DataFrame(b_of_p_scaled, columns = b_of_p.columns)
purch_behavior_scaled = pd.DataFrame(purch_behavior_scaled, columns = purch_behavior.columns)
```

In [32]:
```
display(b_of_p_scaled.head(), purch_behavior_scaled.head())
```

	Pr Cat 1	Pr Cat 2	Pr Cat 3	Pr Cat 4	PropCat 5	PropCat 6	PropCat 7	PropCat 8	PropCat 9	PropCat 10	PropCat 11	PropCat 12	PropCat 13
0	-0.161736	0.217155	-0.031227	-0.072320	0.140449	-0.555525	-0.495454	-0.525782	-0.490823	-0.264365	-0.298493	0.829966	-0.261486
1	0.051111	0.174301	-0.165759	-0.126441	-0.003139	1.532717	-0.358265	-0.420163	-0.262803	-0.264365	0.301536	-0.236478	-0.261486
2	-0.566204	-0.561982	1.577546	-0.462691	-0.672526	0.173827	-0.323929	-0.454785	-0.352876	-0.264365	-0.298493	0.380997	-0.261486
3	-0.994241	-0.299177	1.720601	-0.462691	-0.180850	-0.555525	-0.495454	-0.525782	-0.490823	-0.264365	-0.298493	-0.236478	-0.261486
4	-0.994241	-1.429203	0.020067	3.752024	1.107504	-0.555525	-0.495454	-0.209632	-0.490823	-0.264365	-0.298493	-0.236478	-0.261486

	No. of Brands	Brand Runs	Total Volume	Value	Trans / Brand Runs	Vol/Tran	Pur Vol No Promo - %	Pur Vol Promo 6 %	Pur Vol Other Promo %	Others 999	max_brand_pct
0	-0.403364	0.120173	-0.501007	-0.588594	-0.464084	-0.324562	0.728635	-0.575827	-0.465644	-0.100244	0.020107
1	0.863748	0.890306	0.265360	0.389966	-0.391077	-0.264213	-0.215341	0.463901	-0.241798	0.597393	-0.793987
2	0.863748	2.045506	1.440672	0.694243	-0.352653	-0.194651	0.239133	-0.366156	0.076043	-0.482045	0.611288
3	-1.036920	-1.131294	-1.341436	-1.386401	-0.621624	-0.161137	0.728635	-0.575827	-0.465644	-1.757097	0.802652
4	-0.403364	-0.938760	-0.465587	-0.845841	-0.172058	0.898835	-2.500632	0.980286	2.884545	0.960147	-0.795121

Each DF are scaled and should be ready to utilize in K-means clustering

1) Use K-means clustering to identify clusters of households based on:

Notes to think on and will be written about as each problem is solve:

1 - how should K be chosen? Think about how the clusters would be used. It is likely that the marketing efforts would support two to five different promotional approachs

2 - How should the percentage of total purchases by various brands be treated? Isn't a customer who buys all brand A just as loyal as a customer who buys all brand B? What will be the single effect on any distance measure of using the brand share variables as is? Consider using a single derived variable.

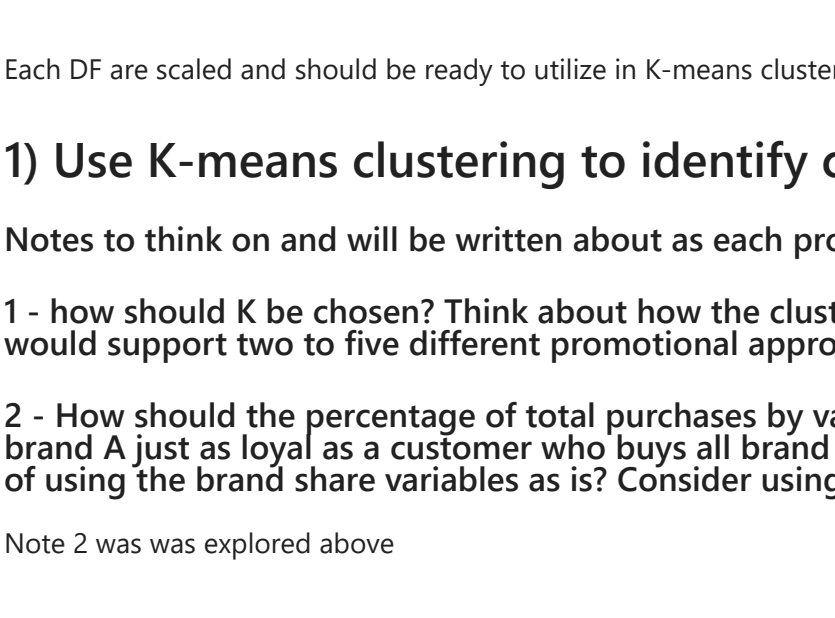
Note 2 was was explored above

1.1 The variables that describe purchase behavior (including brand loyalty) (10 points)

In [33]:
```
# we will utilize KMeans() to run the clustering on purch_behavior_scaled
# before understanding centroids of clusters, let's pick optimal amount of clusters


kmeans_sse_pb = []

# since we can only support two to five different approaches
# discover clusters for each quantity of approach

for x in range(2,6):
    kmeans_pb = KMeans(n_clusters = x, random_state = 0).fit(purch_behavior_scaled)
    kmeans_sse_pb.append(kmeans_pb.inertia_) # to save SSE score and later use to graph and compare results
```

In [34]:
```
# visualizing both the centroids and SSE to see optimal number of clusters

plt.plot(range(2,6), kmeans_sse_pb)
plt.xticks(range(2,6))
plt.xlabel("Number of Clusters")
plt.ylabel("SSE")
plt.title("Number of Clusters vs SSE")
```

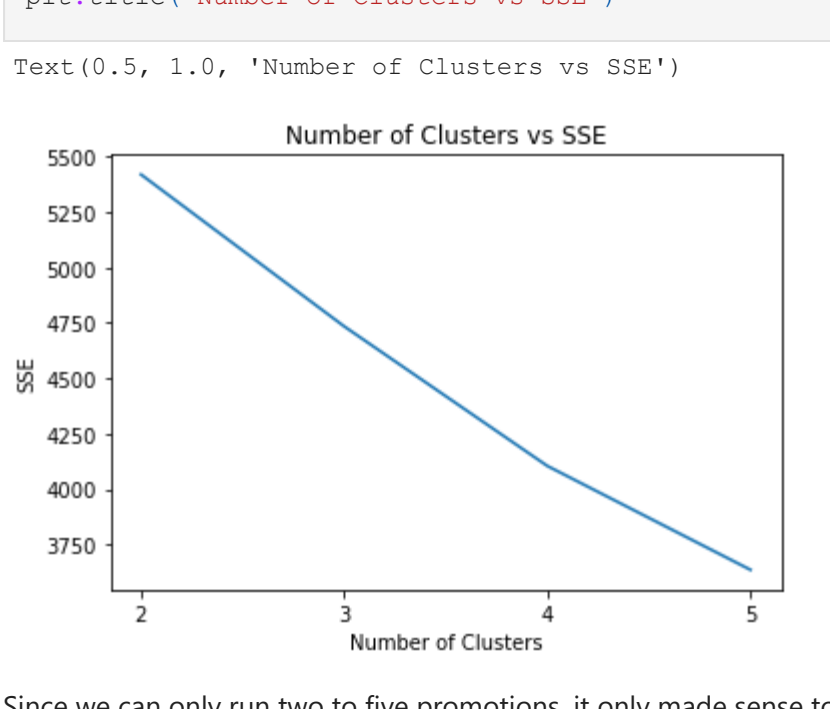
Out[34]: Text(0.5, 1.0, 'Number of Clusters vs SSE')



Since we can only run two to five promotions, it only made sense to view the amount of clusters in this range to begin. However, since it's continuing to decrease and not quite hitting the "elbow" of the elbow chart, perhaps it could be worth to at least explore if we can improve the SSE with more clusters.

In [35]:
```
kmeans_sse_pb_ten = []

# since we can only support two to five different approaches
# discover clusters for each quantity of approach

for x in range(2,11):
    kmeans_pb = KMeans(n_clusters = x, random_state = 0).fit(purch_behavior_scaled)
    kmeans_sse_pb_ten.append(kmeans_pb.inertia_) # to save SSE score and later use to graph and compare results


# visualizing both the centroids and SSE to see optimal number of clusters

plt.plot(range(2,11), kmeans_sse_pb_ten)
plt.xticks(range(2,11))
plt.xlabel("Number of Clusters")
plt.ylabel("SSE")
plt.title("Number of Clusters vs SSE \n using Purchase Behavior data")
```

Out[35]: Text(0.5, 1.0, 'Number of Clusters vs SSE \n using Purchase Behavior data')



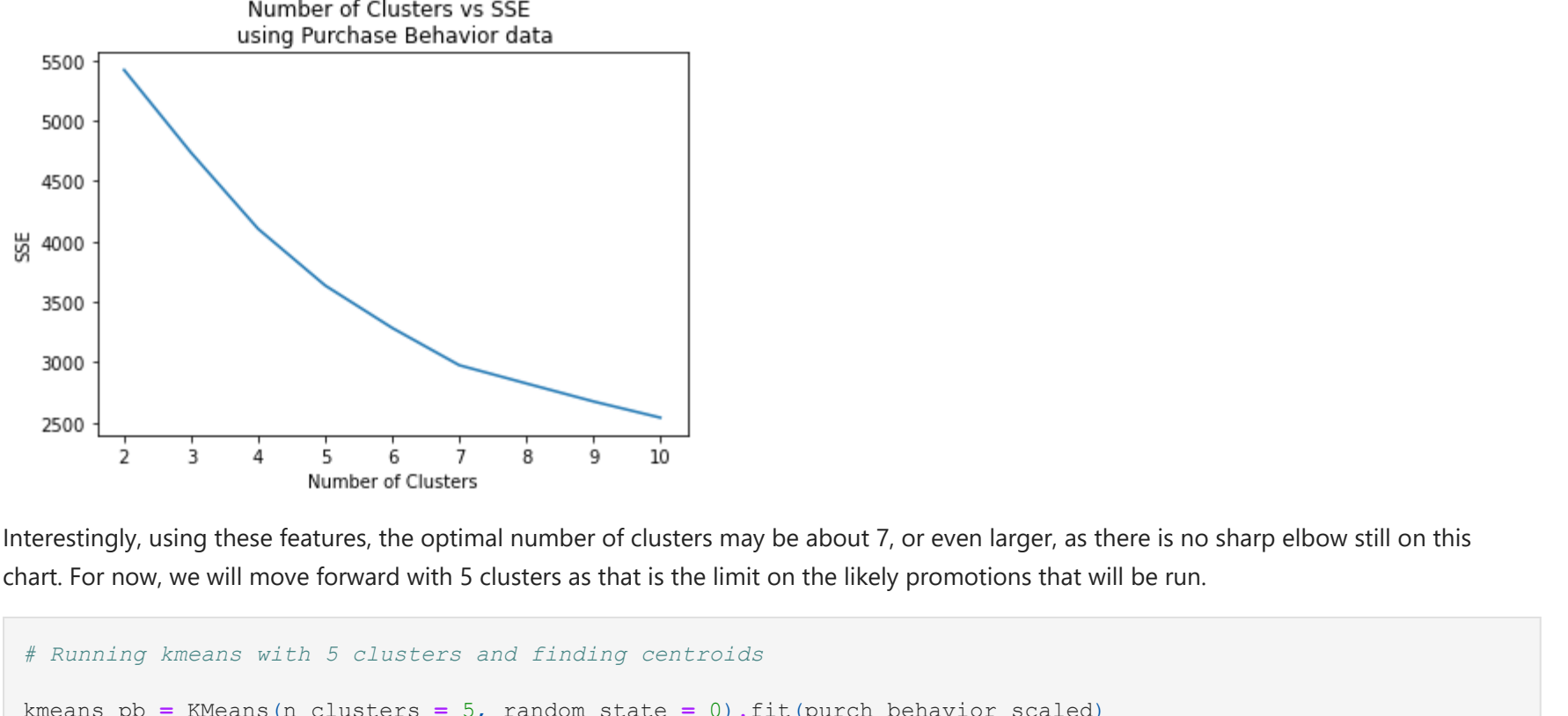
Interestingly, using these features, the optimal number of clusters may be about 7, or even larger, as there is no sharp elbow still on this chart. For now, we will move forward with 5 clusters as that is the limit on the likely promotions that will be run.

In [36]:
```
# Running kmeans with 5 clusters and finding centroids

kmeans_pb = KMeans(n_clusters = 5, random_state = 0).fit(purch_behavior_scaled)
centroids_pb = pd.DataFrame(kmeans_pb.cluster_centers_, index = [1, 2, 3, 4, 5])
centroids_pb.columns = purch_behavior_scaled.columns


centroids_pb
```

Out[36]:

	No. of Brands	Brand Runs	Total Volume	Value	Trans / Brand Runs	Vol/Tran	Pur Vol No Promo - %	Pur Vol Promo 6 %	Pur Vol Other Promo %	Others 999	max_brand_pct
1	-0.639945	-0.414777	-0.501366	-0.441904	-0.133561	-0.181697	0.253098	-0.206108	-0.153881	0.901349	-0.731852
2	-0.009151	0.077388	2.365251	2.180262	0.079408	1.998428	0.176219	-0.251126	0.031883	-0.044344	0.115589
3	-0.512320	-0.773206	-0.214904	-0.389538	0.606049	0.184663	0.417898	-0.431664	-0.136071	-1.175967	1.288097
4	0.979559	0.905315	0.160177	0.281826	-0.315964	-0.378778	0.097886	-0.040607	-0.110028	0.132840	-0.385746
5	-0.004458	0.278835	-0.430990	-0.362095	-0.349095	-0.365939	-2.439558	2.207217	1.198274	0.361131	-0.371325

In [37]:
```
# plotting centroids of each cluster (Profile Plot) based on feature

centroids_pb['cluster'] = centroids_pb.index
plt.figure(figsize=(24,6))
parallel_coordinates(centroids_pb, class_column = 'cluster', colormap= 'Dark2', linewidth=5)
plt.legend(loc='upper left', bbox_to_anchor=(1, 0.5))

plt.title("Profile Plot of KMeans clustering using Purchase Behavior Data")
plt.show()
```



Below are some observations of each cluster:

Cluster 1: relatively below average in most areas - most interestingly, lowest score for max_brand percentage while of course spending the most on other brands -- the least 'brand loyal'

Cluster 2: Very high volume and buys a lot on each purchase. relatively unaffected by promo material

Cluster 3: The most 'brand loyal' as this segment makes the most transactions per brand run and highest percentage of purchases to one single brand

Cluster 4: Spends on a lot of different brands and 2nd most volume -- will go on longest brand runs as well.

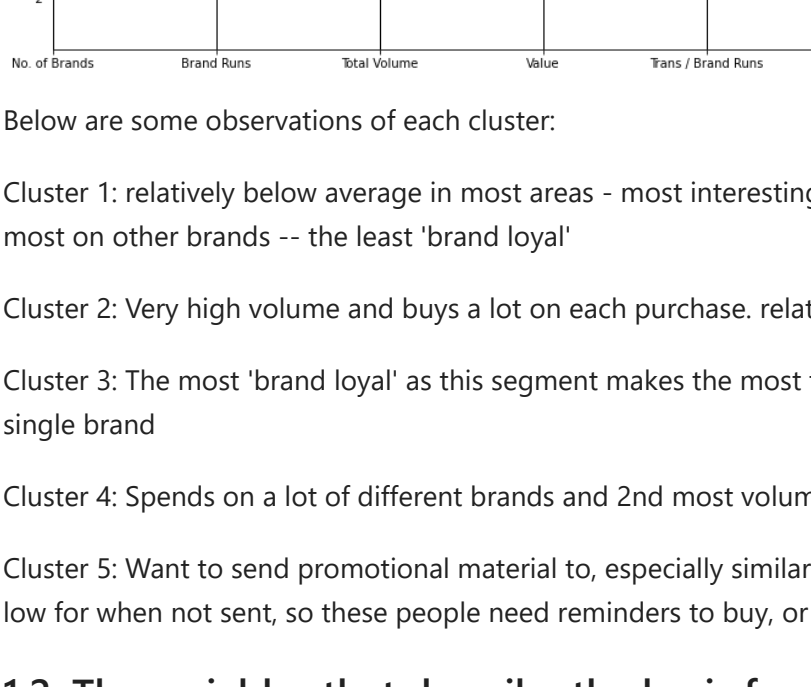
Cluster 5: Want to send promotional material to, especially similar to promo 6. Very high bheavior for when sent promo material and very low for when not sent, so these people need reminders to buy, or only buy because of a sale going on.

1.2. The variables that describe the basis for purchase (10 points)

In [38]:
```
# simlarly to 1.1, we will utilize KMeans() to run the clustering on basis of purchase - b_of_p_scaled
# before understanding centroids of clusters, let's pick optimal amount of clusters


kmeans_sse_bop = []

# since last one continued to decrease after 5, let's see how this looks up to 10
# discover clusters for each quantity of approach

for x in range(2,11):
    kmeans_bop = KMeans(n_clusters = x, random_state = 0).fit(b_of_p_scaled)
    kmeans_sse_bop.append(kmeans_bop.inertia_) # to save SSE score and later use to graph and compare results
```

In [39]:
```
# visualizing both the centroids and SSE to see optimal number of clusters

plt.plot(range(2,11), kmeans_sse_bop)
plt.xticks(range(2,11))
plt.xlabel("Number of Clusters")
plt.ylabel("SSE")
plt.title("Number of Clusters vs SSE \n using Basis of Purchase data")
```

Out[39]: Text(0.5, 1.0, 'Number of Clusters vs SSE \n using Basis of Purchase data')

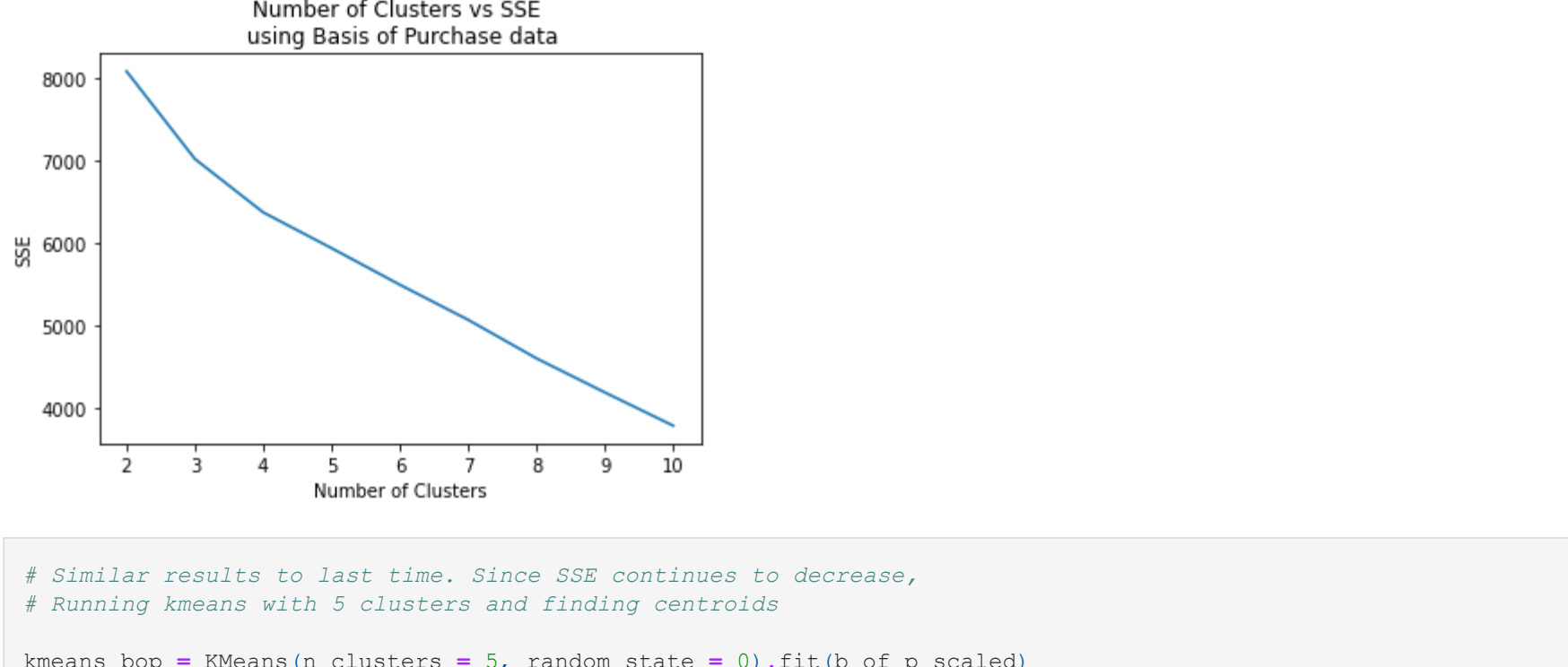


In [40]:
```
# Similar results to last time. Since SSE continues to decrease,
# Running kmeans with 5 clusters and finding centroids

kmeans_bop = KMeans(n_clusters = 5, random_state = 0).fit(b_of_p_scaled)
centroids_bop = pd.DataFrame(kmeans_bop.cluster_centers_, index = [1, 2, 3, 4, 5])
centroids_bop.columns = b_of_p_scaled.columns


centroids_bop
```

Out[40]:

	Pr Cat 1	Pr Cat 2	Pr Cat 3	Pr Cat 4	PropCat 5	PropCat 6	PropCat 7	PropCat 8	PropCat 9	PropCat 10	PropCat 11	PropCat 12	PropCat 13
1	-0.494952	-1.040309	-0.275640	2.801730	0.920385	-0.381518	-0.302817	-0.059867	-0.058768	-0.215105	-0.198017	-0.202545	-0.241008
2	1.774839	-0.963963	-0.463993	-0.385033	-0.350634	0.371439	0.459305	-0.197595	-0.103248	0.762366	-0.177898	0.086277	0.866949
3	-0.362620	0.730560	-0.306242	-0.227944	0.387809	0.039727	0.052607	-0.259617	0.099393	-0.132270	-0.047956	-0.094382	-0.178468
4	0.246892	0.383936	-0.470564	-0.327883	-0.690084	-0.211456	-0.172129	2.071473	-0.044664	-0.096220	0.928400	0.661718	-0.055782
5	-0.790804	-1.131229	2.373219	-0.320848	-1.093152	-0.190072	-0.443118	-0.456981	-0.166758	-0.256810	-0.229474	-0.168121	-0.232433

In [41]:
```
# plotting centroids of each cluster (Profile Plot) based on feature

centroids_bop['cluster'] = centroids_bop.index
plt.figure(figsize=(24,6))
parallel_coordinates(centroids_bop, class_column = 'cluster', colormap= 'Dark2', linewidth=5)
plt.legend(loc='upper left', bbox_to_anchor=(1, 0.5))

plt.title("Profile Plot of KMeans clustering using Basis of Purchase data")
plt.show()
```



Below are some observations on each cluster here:

Off the bat, it seems each cluster has a well-defined spike for each.

Cluster 1: Its spike is in price category is 4 , while having its lowest in category 2, and a low point in selling proposition 6.Also, they spend the second least in price.

Cluster 2: Price category 1 is the most volume for this cluster, while selling proposition categories of 10 and 13 are highs, as well as the highest average price spent.

Cluster 3: price category 2; while relatively average in the rest.

Cluster 4: There are no price categories where it's the most -- but, selling propsoiting 8 is the second highest of all selling proposition for any cluster; it also spends the second most.

Cluster 5: There are two large positive spikes and three large negatives -- they are the highest in price category 3 and selling proposition category 14; however, this cluster has the lowest average price and lowest score for selling proposition with proposition 5.

1.3. The variables that describe both purchase behavior and basis of purchase (14 points)

In [42]:
```
# need to combine datasets. Since they were scaled separately, we should concatenate them together, unscaled, a

all_df = pd.concat([b_of_p, purch_behavior], axis = 1)

all_df.head()
```

Out[42]:

	Pr Cat 1	Pr Cat 2	Pr Cat 3	Pr Cat 4	PropCat 5	PropCat 6	PropCat 7	PropCat 8	PropCat 9	PropCat 10	...	Brand Runs	Total Volume	Value	Trans / Brand Runs	Vol/Tr
0	0.233645	0.560748	0.130841	0.074766	0.501558	0.000000	0.000000	0.000000	0.000000	0.0	...	17	8025	818.0	1.41	334.
1	0.293381	0.547406	0.094812	0.064401	0.456172	0.347048	0.026834	0.016100	0.014311	0.0	...	25	13975	1681.5	1.60	349.
2	0.120130	0.318182	0.561688	0.000000	0.244589	0.121212	0.033550	0.010823	0.008658	0.0	...	37	23100	1950.0	1.70	366.
3	0.000000	0.400000	0.600000	0.000000	0.400000	0.000000	0.000000	0.000000	0.000000	0.0	...	4	1500	114.0	1.00	375.
4	0.000000	0.048193	0.144578	0.807229	0.807229	0.000000	0.000000	0.048193	0.000000	0.0	...	6	8300	591.0	2.17	638.

5 rows × 27 columns

In [43]:
```
# now that they are together, let's scale

scaler_all = StandardScaler()

scaler_all.fit(all_df)

all_scaled = scaler_all.transform(all_df)

all_scaled = pd.DataFrame(all_scaled, columns = all_df.columns)

all_scaled.head()
```

Out[43]:

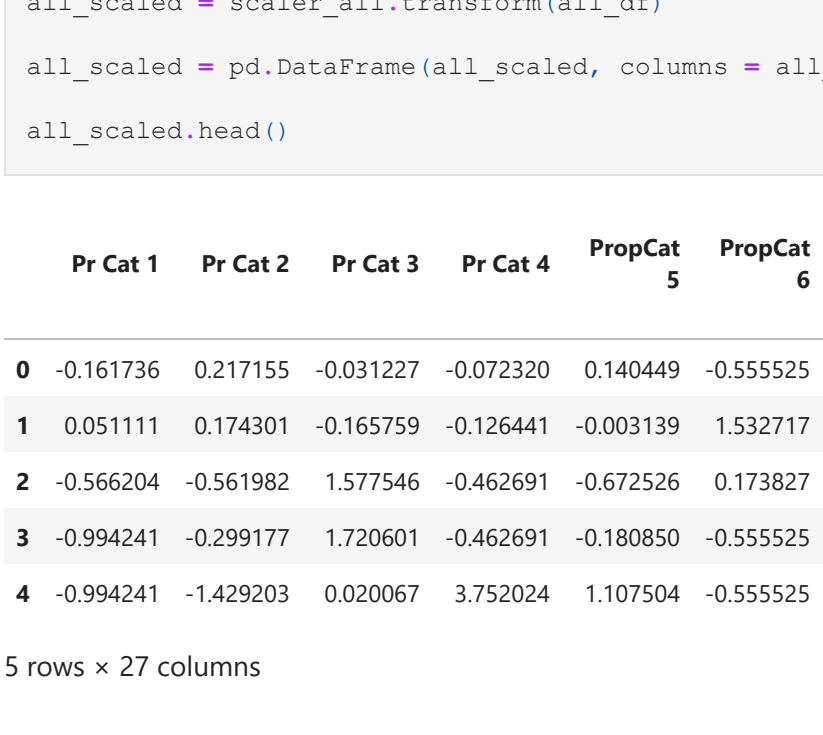
	Pr Cat 1	Pr Cat 2	Pr Cat 3	Pr Cat 4	PropCat 5	PropCat 6	PropCat 7	PropCat 8	PropCat 9	PropCat 10	...	Brand Runs	Total Volume	Valu
0	-0.161736	0.217155	-0.031227	-0.072320	0.140449	-0.555525	-0.495454	-0.525782	-0.490823	-0.264365	...	0.120173	-0.501007	-0.58859
1	0.051111	0.174301	-0.165759	-0.126441	-0.003139	1.532717	-0.358265	-0.420163	-0.262803	-0.264365	...	0.890306	0.265360	0.38996
2	-0.566204	-0.561982	1.577546	-0.462691	-0.672526	0.173827	-0.323929	-0.454785	-0.352876	-0.264365	...	2.045506	1.440672	0.69424
3	-0.994241	-0.299177	1.720601	-0.462691	-0.180850	-0.555525	-0.495454	-0.525782	-0.490823	-0.264365	...	-1.131294	-1.341436	-1.38640
4	-0.994241	-1.429203	0.020067	3.752024	1.107504	-0.555525	-0.495454	-0.209632	-0.490823	-0.264365	...	-0.938760	-0.465587	-0.84584

5 rows × 27 columns

In [44]:
```
# simlarly to the first two, we will utilize KMeans() to run the clustering on both datasets combined
# before understanding centroids of clusters, let's pick optimal amount of clusters


kmeans_sse_all  = []

# since last one continued to decrease after 5, let's see how this looks up to 10
# discover clusters for each quantity of approach

for x in range(2,11):
    kmeans_all = KMeans(n_clusters = x, random_state = 0).fit(all_scaled)
    kmeans_sse_all.append(kmeans_all.inertia_) # to save SSE score and later use to graph and compare results
```

In [45]:
```
# visualizing both the centroids and SSE to see optimal number of clusters

plt.plot(range(2,11), kmeans_sse_all)
plt.xticks(range(2,11))
plt.xlabel("Number of Clusters")
plt.ylabel("SSE")
plt.title("Number of Clusters vs SSE \n using all data (except demographic)")
```

Out[45]: Text(0.5, 1.0, 'Number of Clusters vs SSE \n using all data (except demographic)')



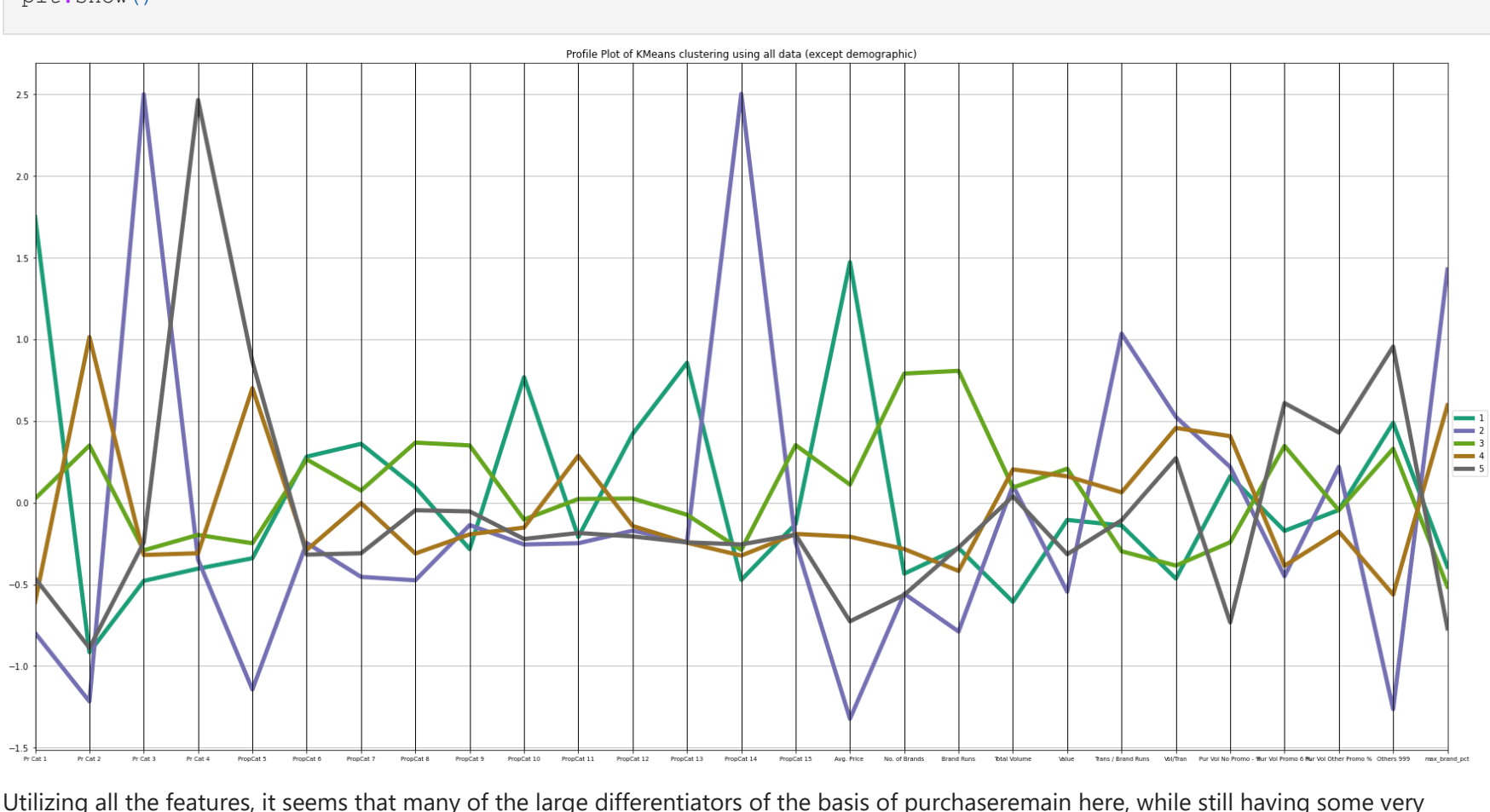
In [46]:
```
# Similar results to last time. Since SSE continues to decrease,
# Running kmeans with 5 clusters and finding centroids

kmeans_all = KMeans(n_clusters = 5, random_state = 0).fit(all_scaled)
centroids_all = pd.DataFrame(kmeans_all.cluster_centers_, index = [1, 2, 3, 4, 5])
centroids_all.columns = all_scaled.columns

centroids_all
```

Out[46]:

	Pr Cat 1	Pr Cat 2	Pr Cat 3	Pr Cat 4	PropCat 5	PropCat 6	PropCat 7	PropCat 8	PropCat 9	PropCat 10	...	Brand Runs	Total Volume	Valu
1	1.748679	-0.915847	-0.479035	-0.403879	-0.340297	0.281673	0.360201	0.094318	-0.284436	0.768220	...	-0.278238	-0.607064	-0.10568
2	-0.797792	-1.216931	2.500125	-0.348753	-1.143950	-0.246906	-0.454436	-0.475243	-0.137204	-0.255947	...	-0.788859	0.105076	-0.54545
3	0.024666	0.349912	-0.291566	-0.197230	-0.248484	0.265312	0.074333	0.367769	0.350417	-0.103390	...	0.807061	0.092180	0.20803
4	-0.608162	1.014162	-0.320122	-0.309728	0.701159	-0.289704	-0.002646	-0.311120	-0.193915	-0.152618	...	-0.418073	0.203787	0.16151
5	-0.463163	-0.888853	-0.244011	2.464729	0.863468	-0.317812	-0.310098	-0.046339	-0.053516	-0.222316	...	-0.274062	0.039283	-0.31567

5 rows × 27 columns

In [47]:
```
# plotting centroids of each cluster (Profile Plot) based on feature

centroids_all['cluster'] = centroids_all.index
plt.figure(figsize=(30,15))
parallel_coordinates(centroids_all, class_column = 'cluster', colormap= 'Dark2', linewidth=5)
plt.legend(loc='upper left', bbox_to_anchor=(1, 0.5))
plt.xticks(fontsize = 7)

plt.title("Profile Plot of KMeans clustering using all data (except demographic)")
plt.show()
```



Utilizing all the features, it seems that many of the large differentiators of the basis of purchaseremain here, while still having some very practical data from the purchase behavior -- this seems to have the best overall segmentation.

2) Select what you think is the best segmentation and comment on the characteristics (demographic, brand loyalty, and basis for purchase) of these clusters).

(This information would be used to guide the development of advertising and promotional campaigns) (14 points)

In continuation of the last answer, I believe the best segmentation would be with all the features. It captures many sides of the buyer -- how they spend and buy, how they respond to promotions, and how loyal they are to brands.

Looking at the Profile Plot of the dataframe using all features (except demographic), it's again apparent that there are some large differences between these 5 clusters. Below will be observations of each:

Cluster 1: High in price category 1, while also high in selling category 10 and 13; has the highest average price of purchases, and is not very loyal to brands. Doesn't buy very often, but when they do, its a lot.

Cluster 2: High in price category 3, and very low in 2 and 5. High in selling prop category 14, yet a very low average per purchase. They are the most brand loyal and they also make the most transactions per brand run.

Cluster 3: They use the most different brands, and are the 2nd least 'brand loyal', based on max percentage of a single brand. They have a decently high response rate to promo #6.

Cluster 4: Price category 2 and selling prop 5 are its two high spikes. It seems they are also more likely to spend if not promoted to and less if promoted to. They are the second most brand loyal.

Cluster 5: Price category 4 and the least brand loyal. They are also the most responsive to promotional material, whether its promo #6 or other, and they spend less when not promoted to.

3) Develop a model that classifies the data into these segments. Since this information would most likely be used in targeting direct-mail promotions, it would be useful to select a marketing segment that would be defined as a success in the classification model. (14 points).

In [56]:
```
# saving the cluster we will be segmenting each customer into in a new column
df = pd.read_csv("C:/Users/steph/OneDrive/Documents/USD/ADS505/Data/BathSoapHousehold.csv")

labels_all = kmeans_all.labels_.tolist()

df['segment'] = labels_all

df.head()
```

Out[56]:

	Member id	SEC	FEH	MT	SEX	AGE	EDU	HS	CHILD	CS	...	PropCat 7	PropCat 8	PropCat 9	PropCat 10	PropCat 11	PropCat 12	PropCat 13	PropCa 14
0	1010010	4	3	10	1	4	4	2	4	1	...	0.000000	0.000000	0.000000	0.0	0.000000	0.028037	0.0	0.13084
1	1010020	3	2	10	2	2	4	4	2	1	...	0.026834	0.016100	0.014311	0.0	0.059034	0.000000	0.0	0.08050
2	1014020	2	3	10	2	4	5	6	4	1	...	0.033550	0.010823	0.008658	0.0	0.000000	0.016234	0.0	0.56168
3	1014030	4	0	0	0	4	0	0	5	0	...	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.0	0.60000
4	1014190	4	1	10	2	3	4	4	3	1	...	0.000000	0.048193	0.000000	0.0	0.000000	0.000000	0.0	0.14457

5 rows × 47 columns

The segments/clusters of each are now back in the original dataframe. We want to predict for the segment they are in for new clients to market to them in the most optimized way.

Before building the model, we will need to clean the data once again in a similar fashion as before for it to be ready to model with. We will also be runnning multi-nomial classifications with logistic regression, KNN, decision tree, and random forest.

In [57]:
```
# getting data ready to model with
# dropping member id and soon-to-be dummmied variables

df_final = df.drop(['Member id', 'FEH', 'MT', 'AGE', 'EDU', 'CHILD', 'segment'], axis = 1)

y = df['segment']

# 'FEH', 'MT', 'AGE', 'EDU', 'CHILD' - all could be categorical variables based on how bucketed / categorized
feh_dummy = pd.get_dummies(df['FEH'], prefix = 'feh', drop_first = True)
mt_dummy = pd.get_dummies(df['MT'], prefix = 'mt', drop_first = True)
age_dummy = pd.get_dummies(df['AGE'], prefix = 'age', drop_first = True)
edu_dummy = pd.get_dummies(df['EDU'], prefix = 'edu', drop_first = True)
child_dummy = pd.get_dummies(df['CHILD'], prefix = 'child', drop_first = True)

# bringing these back into dataframe

df_final = pd.concat([df_final, feh_dummy, mt_dummy, age_dummy, edu_dummy, child_dummy], axis = 1)

df_final.head()
```

Out[57]:

	SEC	SEX	HS	CS	Affluence Index	No. of Brands	Brand Runs	Total Volume	No. of Trans	Value	...	edu_4	edu_5	edu_6	edu_7	edu_8	edu_9	child_2	child_3	chi
0	4	1	2	1	2	3	17	8025	24	818.0	...	1	0	0	0	0	0	0	0	
1	3	2	4	1	19	5	25	13975	40	1681.5	...	1	0	0	0	0	0	1	0	
2	2	2	6	1	23	5	37	23100	63	1950.0	...	0	1	0	0	0	0	0	0	
3	4	0	0	0	0	2	4	1500	4	114.0	...	0	0	0	0	0	0	0	0	
4	4	2	4	1	10	3	6	8300	13	591.0	...	1	0	0	0	0	0	0	1	

5 rows × 73 columns

In [58]:
```
# Now to scale these features

scaler = StandardScaler()

scaler.fit(df_final)

df_final_scaled = scaler.transform(df_final)

df_final_scaled = pd.DataFrame(df_final_scaled, columns = df_final.columns)

df_final_scaled.head()
```

Out[58]:

	SEC	SEX	HS	CS	Affluence Index	No. of Brands	Brand Runs	Total Volume	No. of Trans	Value	...	edu_4	edu_5	edu_
0	1.341641	-1.139458	-0.953656	0.134793	-1.317478	-0.403364	0.120173	-0.501007	-0.410811	-0.588594	...	1.847096	-0.678125	-0.19965
1	0.447214	0.403826	-0.083400	0.134793	0.173676	0.863748	0.890306	0.265360	0.508057	0.389966	...	1.847096	-0.678125	-0.19965
2	-0.447214	0.403826	0.786857	0.134793	0.524535	0.863748	2.045506	1.440672	1.828930	0.694243	...	-0.541390	1.474654	-0.19965
3	1.341641	-2.682741	-1.823913	-1.837792	-1.492908	-1.036920	-1.131294	-1.341436	-1.559396	-1.386401	...	-0.541390	-0.678125	-0.19965
4	1.341641	0.403826	-0.083400	0.134793	-0.615759	-0.403364	-0.938760	-0.465587	-1.042532	-0.845841	...	1.847096	-0.678125	-0.19965

5 rows × 73 columns

In [59]:
```
# how large is the dataset again?

df_final_scaled.shape
```

Out[59]:
```
(600, 73)
```

In [60]:
```
# now that the data is set, let's split into train, validation, and test set
# since there are only 600 points, let's use cross validation with training set and hold out just the test set

X = df_final_scaled

# 80 percent train, 20 percent test (480 vs 120 records)

trainX, testX, trainy, testy = train_test_split(X, y, test_size = .2, random_state = 12345)
```

In [81]:
```
# first logistic regression

# importing for stratified KFold test
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression

# define the multinomial logistic regression model
logit = LogisticRegression(multi_class='multinomial', solver='lbfgs')

# define the model evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

# evaluate the model and collect the scores
n_scores_logit = cross_val_score(logit, trainX, trainy, scoring='accuracy', cv=cv, n_jobs=-1)

# report the model performance
print('Mean Accuracy: %.3f (%.3f)' % (np.mean(n_scores), np.std(n_scores)))
```

```
Mean Accuracy: 0.919 (0.035)
```

In [82]:
```
# fitting logistic regression

logit.fit(trainX, trainy)

# seeing confusion matrix

classificationSummary(testy, logit.predict(testX))
```

```
Confusion Matrix (Accuracy 0.8750)

       Prediction
Actual  0  1  2  3  4
     0 16  0  5  2  0
     1  0  8  0  0  0
     2  3  0 37  4  0
     3  0  0  1 31  0
     4  0  0  0  0 13
```

In [87]:
```
# saving scores into a dictionary

from sklearn.metrics import accuracy_score, precision_score, recall_score

logit_results = {'Model': 'Logistic regression', 'Accuracy': accuracy_score(testy, logit.predict(testX)),
                 'Precision': precision_score(testy, logit.predict(testX), average = 'weighted'),
                 'Recall': recall_score(testy, logit.predict(testX), average = 'weighted')}

logit_results
```

Out[87]:
```
{'Model': 'Logistic regression',
 'Accuracy': 0.875,
 'Precision': 0.8753308081643455,
 'Recall': 0.875}
```

In [71]:
```
# Kneighbors
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(algorithm = 'auto')

parameters_KNN = {
    'n_neighbors': (1,10, 1),
    'leaf_size': (20,40,1),
    'p': (1,2),
    'weights': ('uniform', 'distance'),
    'metric': ('minkowski', 'chebyshev')}

grid_search_KNN = GridSearchCV( estimator=knn,
    param_grid=parameters_KNN,
    scoring = 'accuracy',
    n_jobs = -1,
    cv = 5)
```

In [74]:
```
knn=grid_search_KNN.fit(trainX, trainy)

#Parameter setting that gave the best results on the hold out data.
print(grid_search_KNN.best_params_ )
#Mean cross-validated score of the best_estimator
print('Best Score - KNN:', grid_search_KNN.best_score_ )
```

```
{'leaf_size': 20, 'metric': 'minkowski', 'n_neighbors': 10, 'p': 1, 'weights': 'distance'}
Best Score - KNN: 0.8208333333333334
```

In [79]:
```
knn = KNeighborsClassifier(n_neighbors = 10, leaf_size = 20, metric = 'minkowski', weights = 'distance', algori

knn.fit(trainX, trainy)

# seeing confusion matrix of classifications

classificationSummary(testy, knn.predict(testX))
```

```
Confusion Matrix (Accuracy 0.7167)

       Prediction
Actual  0  1  2  3  4
     0 12  0 10  1  0
     1  0  8  0  0  0
     2  4  1 31  8  0
     3  0  0  3 29  0
     4  0  1  4  2  6
```

In [88]:
```
knn_results = {'Model': 'KNN', 'Accuracy': accuracy_score(testy, knn.predict(testX)),
               'Precision': precision_score(testy, knn.predict(testX), average = 'weighted'),
               'Recall': recall_score(testy, knn.predict(testX), average = 'weighted')}

knn_results
```

Out[88]:
```
{'Model': 'KNN',
 'Accuracy': 0.7166666666666667,
 'Precision': 0.7355555555555555,
 'Recall': 0.7166666666666667}
```

In [92]:
```
# Decision Tree

dt = DecisionTreeClassifier(random_state = 0)

parameters_dt = {
    'max_depth': range(1,10,1),
    'criterion': ('gini', 'entropy')}

grid_search_dt = GridSearchCV( estimator=dt,
    param_grid= parameters_dt,
    scoring = 'accuracy',
    n_jobs = -1,
    cv = 5)
```

In [93]:
```
dt = grid_search_dt.fit(trainX, trainy)

#Parameter setting that gave the best results on the hold out data.
print(grid_search_dt.best_params_ )
#Mean cross-validated score of the best_estimator
print('Best Score - Decision Tree:', grid_search_dt.best_score_ )
```

```
{'criterion': 'gini', 'max_depth': 8}
Best Score - Decision Tree: 0.83125
```

In [95]:
```
# fitting with optimal parameters

dt = DecisionTreeClassifier(criterion = 'gini', max_depth = 8, random_state = 0)

dt.fit(trainX, trainy)

# seeing confusion matrix of classifications

classificationSummary(testy, dt.predict(testX))
```

```
Confusion Matrix (Accuracy 0.7667)

       Prediction
Actual  0  1  2  3  4
     0 18  0  3  2  0
     1  0  7  1  0  0
     2  3  0 35  5  1
     3  2  0  9 21  0
     4  1  0  1  0 11
```

In [96]:
```
dt_results = {'Model': 'Decision Tree', 'Accuracy': accuracy_score(testy, dt.predict(testX)),
              'Precision': precision_score(testy, dt.predict(testX), average = 'weighted'),
              'Recall': recall_score(testy, dt.predict(testX), average = 'weighted')}

dt_results
```

Out[96]:
```
{'Model': 'Decision Tree',
 'Accuracy': 0.7666666666666667,
 'Precision': 0.7716269841269842,
 'Recall': 0.7666666666666667}
```

In [99]:
```
# finally, random forest

rf = RandomForestClassifier(random_state = 0)

parameters_rf = { 'n_estimators': range(1, 600, 100),
    'max_depth': range(1,12,1),
    'criterion': ('gini', 'entropy', 'log_loss')}

grid_search_rf = GridSearchCV( estimator=rf,
    param_grid= parameters_rf,
    scoring = 'accuracy',
    n_jobs = -1,
    cv = 5)
```

In [100…
```
rf = grid_search_rf.fit(trainX, trainy)

#Parameter setting that gave the best results on the hold out data.
print(grid_search_rf.best_params_ )
#Mean cross-validated score of the best_estimator
print('Best Score - Random Forest:', grid_search_rf.best_score_ )
```

```
{'criterion': 'gini', 'max_depth': 11, 'n_estimators': 401}
Best Score - Random Forest: 0.8979166666666668
```

In [104…
```
# fitting with optimal parameters

rf = RandomForestClassifier(criterion = 'gini', max_depth = 11, n_estimators = 401, random_state = 0)

rf.fit(trainX, trainy)

# seeing confusion matrix of classifications

classificationSummary(testy, rf.predict(testX))
```

```
Confusion Matrix (Accuracy 0.9083)

       Prediction
Actual  0  1  2  3  4
     0 17  0  4  2  0
     1  0  8  0  0  0
     2  0  0 42  2  0
     3  0  0  1 31  0
     4  1  0  1  0 11
```

In [105…
```
rf_results = {'Model': 'Random Forest', 'Accuracy': accuracy_score(testy, rf.predict(testX)),
              'Precision': precision_score(testy, rf.predict(testX), average = 'weighted'),
              'Recall': recall_score(testy, rf.predict(testX), average = 'weighted')}

rf_results
```

Out[105…
```
{'Model': 'Random Forest',
 'Accuracy': 0.9083333333333333,
 'Precision': 0.9130423280423281,
 'Recall': 0.9083333333333333}
```

In [109…
```
results = [logit_results, knn_results, dt_results, rf_results]

results_df = pd.DataFrame(results)

results_df
```

Out[109…

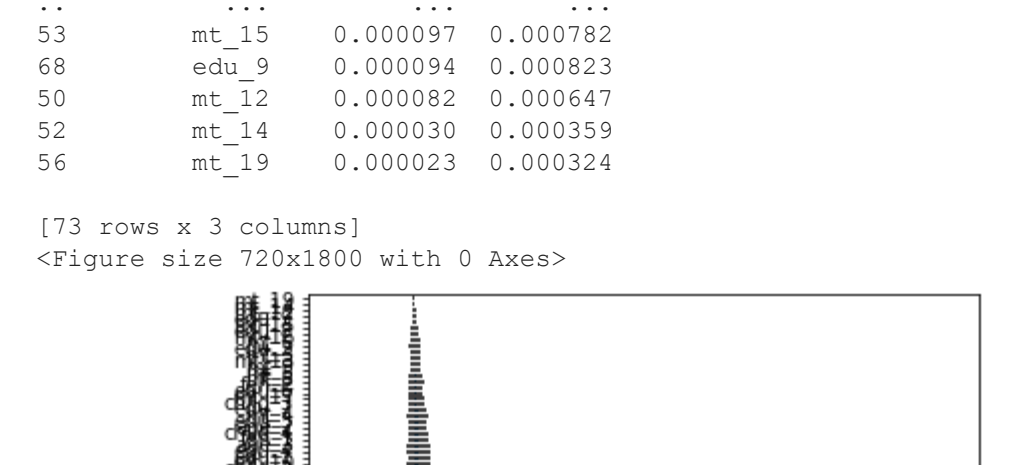
	Model	Accuracy	Precision	Recall
0	Logistic regression	0.875000	0.875331	0.875000
1	KNN	0.716667	0.735556	0.716667
2	Decision Tree	0.766667	0.771627	0.766667
3	Random Forest	0.908333	0.913042	0.908333

Overall, it seemed the random forest had the best results (hyperparameters of gini criterion, maxdepth of 11, and 401 estimators), with logistic regression in second with a respectable accuracy of 87.5%.

Let's take a closer look at the random forest model and the importance of the variables

In [118…
```
# variable (feature) importance plot
importances = rf.feature_importances_
std = np.std([tree.feature_importances_ for tree in rf.estimators_], axis=0)
rf_df = pd.DataFrame({'feature': trainX.columns, 'importance': importances, 'std': std})
rf_df = rf_df.sort_values('importance', ascending = False)
print(rf_df)


fig = plt.figure()
fig.set_figwidth(10)
fig.set_figheight(25)

ax = rf_df.plot(kind='barh', xerr='std', x='feature', legend=False)
ax.set_ylabel('Feature')
plt.show()
```

```
       feature  importance       std
26     Pr Cat 2    0.095268  0.073722
25     Pr Cat 1    0.090339  0.071263
12   Avg. Price    0.073799  0.067645
28     Pr Cat 4    0.061671  0.050692
17   Br. Cd. 55    0.053883  0.074455
..          ...         ...       ...
53        mt_15    0.000097  0.000782
68        edu_9    0.000094  0.000823
50        mt_12    0.000082  0.000647
52        mt_14    0.000030  0.000359
56        mt_19    0.000023  0.000324

[73 rows x 3 columns]
<Figure size 720x1800 with 0 Axes>
```



The top 5 most important features, in descending order, were price category 2, price category 1, average price, price category 4, and brand 55.

This makes sense to a degree as many of the clusters that were identified seemed to differentiate on these features. This is a great starting point to understand the client base more, and have different marketing campaigns tied to each segment.

4) Briefly explain, in two to three paragraphs, the business objective, the data mining models used, why they were used, the model results, and your recommendations to your non-technical stakeholder team. (18 points)

While the company CRISA has segmented its customer market in the past, it hasn't always been as effective as their leadership has wanted as their past promotional cycles haven't been profitable, nor were they close to their target revenue goals. These past segmentations were based solely on demographic information, and today, the company wants to utilize the other data they've collected on their customers to see if there can be more effective promotional strategies. This other data includes information on customer purchase behavior, including volume and frequency of purchasing, as well as levels of brand loyalty, and basis of purchase -- the price and selling proposition on items bought. CRISA can only support up to 5 marketing promotions at once, so it's necessary to keep the segmentations within that range.

While utilizing data mining models, it's possible to segment these need to give marketing leaders the needed context and full-picture of how the clients act and function based on their purchasing habits, responses to promotional material, brand loyalty and more. While the goal would be to create an optimized marketing strategy that maximizes revenue while minimizing spend, another important piece is to understand this customer engagement to possibly create new, innovative ways to engage with clients and truly give a great brand experience.

The purchase behavior of clients and the basis of purchase were each used, as well as used together, in a K means clustering model to segment the clients. When tuning the models, all three also weren't at their most optimal even past 5 clusters; within our constraints, 5 clusters offered the least intertia for all datasets. After analyzing the three results, it seemed the segments of utilizing them all together both gave well differentiated segments and good informative takeaways on what traits each segment have.

For example, Cluster 5 is the least brand loyal (based on max percentage of spend on a given brand), and they are also the most responsive to promotional material, whether its promo #6 or other, and they spend less when not promoted to. Thus, it could be smart to target them with promotional material regardless of the brand, as it seems that is what will drive them to the store. Cluster 2, for instance, spend the most in price category 3, and they are the most brand loyal while they also make the most transactions per brand run. They do, however, spend the least per transaction. Perhaps, the marketing team could target them with their favorite brands with sales that get those prices into category '3', especially right after they buy one thing from a new brand (start a new 'brand run').

Utilizing these new segmentations, a few models were developed to predict which segment new clients could fall into. An ensemble method -- Random Forest -- was ultimately the best model as it performed the multi-class classificationcli, with an accuracy, precision and recall score of over 90%. Seeing the most important variables into predicting the segment, were price category 1, 2 and average price, which seems our segments are somewhat dependent on those. The metrics were very promising, however, and I think it would be beneficial to utilize this model at this point forward, while iterating and perhaps increasing performance over time.

Overall, I think utilizing the results of the segmentation and subsequent supervised modeling would be extremely beneficial to the company. From the cluster results, a strategy can quickly be put into place for each client segment while we can predict where newer customers can fall to decide which strategy to run on them. However, it is important to continue research and iterating on these models. Perhaps the final model can be modified to better understand the clusters from a promotional standpoint, or the clustering can be done with some demographical information in there alongside some of the purchasing behavior. For now, this is a great starting point that can optimize their promotional strategies and perhaps drive better customer experiences as well.