Customer Segmentation with KNN

In [1]:

```
# Importing our libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder
%matplotlib inline
```

In [2]:

```
# Loading our data
train = pd.read_csv('train.csv')
train.head(10)
```

Out[2]:

	ID	Gender	Ever_Married	Age	Graduated	Profession	Work_Experience	Spending_S
0	462809	Male	No	22	No	Healthcare	1.0	
1	462643	Female	Yes	38	Yes	Engineer	NaN	Ave
2	466315	Female	Yes	67	Yes	Engineer	1.0	
3	461735	Male	Yes	67	Yes	Lawyer	0.0	
4	462669	Female	Yes	40	Yes	Entertainment	NaN	
5	461319	Male	Yes	56	No	Artist	0.0	Ave
6	460156	Male	No	32	Yes	Healthcare	1.0	
7	464347	Female	No	33	Yes	Healthcare	1.0	
8	465015	Female	Yes	61	Yes	Engineer	0.0	
9	465176	Female	Yes	55	Yes	Artist	1.0	Ave

In [3]:

```
# #Changing our target variable to numeric
# mappings = {'A':1, 'B':2, 'C':3, 'D':4}
# train['Segmentation'] = train['Segmentation'].map(mappings)
```

In [4]:

```
# Checking the first few rows
train.head()
```

Out[4]:

	ID	Gender	Ever_Married	Age	Graduated	Profession	Work_Experience	Spending_S
0	462809	Male	No	22	No	Healthcare	1.0	
1	462643	Female	Yes	38	Yes	Engineer	NaN	Ave
2	466315	Female	Yes	67	Yes	Engineer	1.0	
3	461735	Male	Yes	67	Yes	Lawyer	0.0	
4	462669	Female	Yes	40	Yes	Entertainment	NaN	

In [5]:

```
# Droppping rows that are not needed
train = train.drop(['ID', 'Var_1'], axis = 1)
```

Getting more insight about our data

In [6]:

```
train.describe()
```

Out[6]:

	Age	Work_Experience	Family_Size
count	8068.000000	7239.000000	7733.000000
mean	43.466906	2.641663	2.850123
std	16.711696	3.406763	1.531413
min	18.000000	0.000000	1.000000
25%	30.000000	0.000000	2.000000
50%	40.000000	1.000000	3.000000
75%	53.000000	4.000000	4.000000
max	89.000000	14.000000	9.000000

In [7]:

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8068 entries, 0 to 8067
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Gender           8068 non-null   object 
 1   Ever_Married     7928 non-null   object 
 2   Age              8068 non-null   int64  
 3   Graduated        7990 non-null   object 
 4   Profession       7944 non-null   object 
 5   Work_Experience  7239 non-null   float64
 6   Spending_Score   8068 non-null   object 
 7   Family_Size      7733 non-null   float64
 8   Segmentation     8068 non-null   object 
dtypes: float64(2), int64(1), object(6)
memory usage: 567.4+ KB
```

In [8]:

```
train.Ever_Married.unique()
```

Out[8]:

```
array(['No', 'Yes', nan], dtype=object)
```

In [9]:

```
train.Ever_Married.value_counts()
```

Out[9]:

```
Yes    4643
No     3285
Name: Ever_Married, dtype: int64
```

In [10]:

```
train['Graduated'].unique()
```

Out[10]:

```
array(['No', 'Yes', nan], dtype=object)
```

In [11]:

```
train['Graduated'].value_counts()
```

Out[11]:

```
Yes    4968
No     3022
Name: Graduated, dtype: int64
```

In [12]:

```
#  The % of missing values in Graduated column
train['Graduated'].isnull().sum() / train.shape[0] * 100
```

Out[12]:

```
0.9667823500247893
```

In [13]:

```
#  The % of missing values in Ever_Married column
train['Ever_Married'].isnull().sum() / train.shape[0] * 100
```

Out[13]:

```
1.7352503718393655
```

In [14]:

```
# The number of entries in our dataset
total_entries = train.shape[0]
total_entries
```

Out[14]:

```
8068
```

In [15]:

```
# The number of missing values in our dataset
train.isnull().sum()
```

Out[15]:

```
Gender              0
Ever_Married      140
Age                 0
Graduated          78
Profession        124
Work_Experience   829
Spending_Score      0
Family_Size       335
Segmentation        0
dtype: int64
```

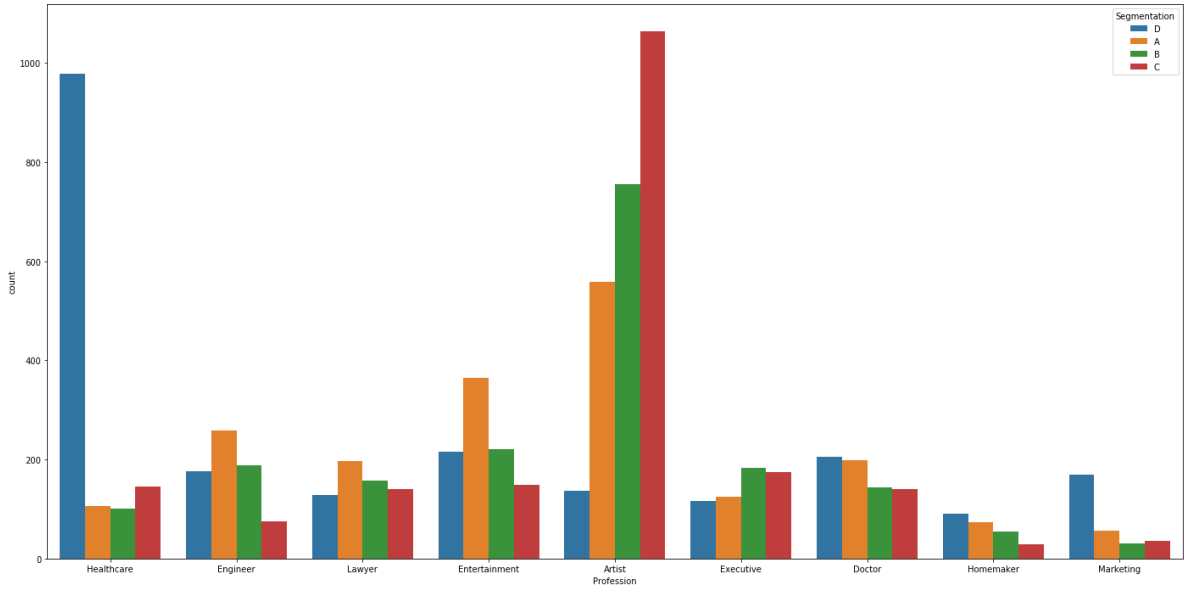
Visuals

In [16]:

```
# Grouping our segments according to profession
plt.rcParams['figure.figsize'] = (24, 12)
sns.countplot('Profession', hue = 'Segmentation', data = train)
```

Out[16]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x24132856a88>
```

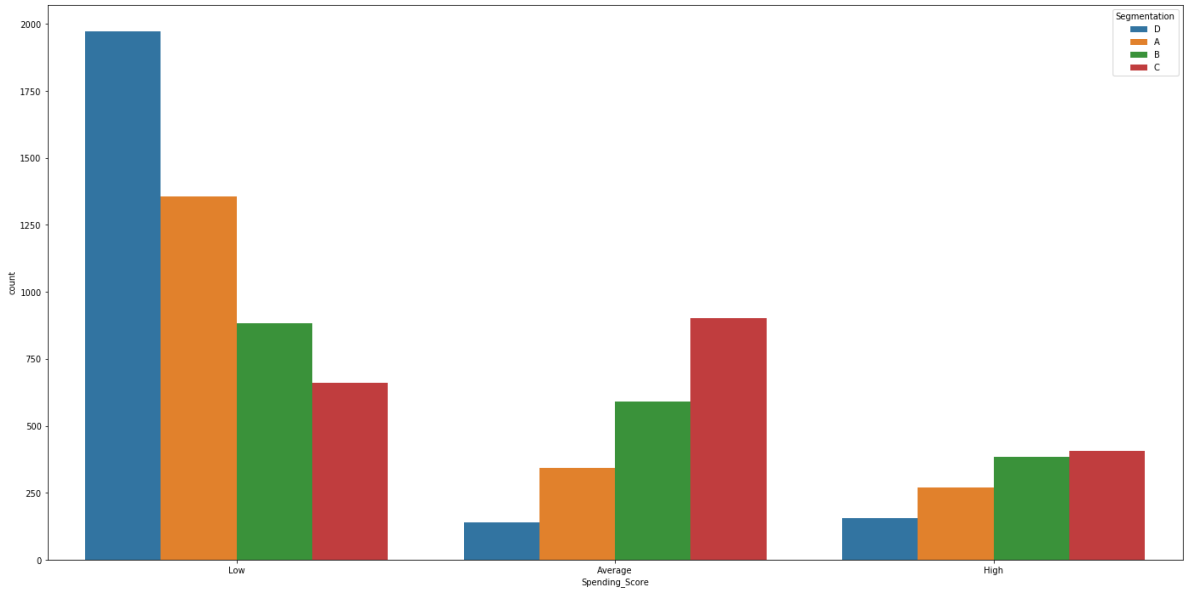


In [17]:

```
# Grouping our segments according to Spending score
sns.countplot('Spending_Score', hue = 'Segmentation', data = train)
```

Out[17]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2413571a048>
```

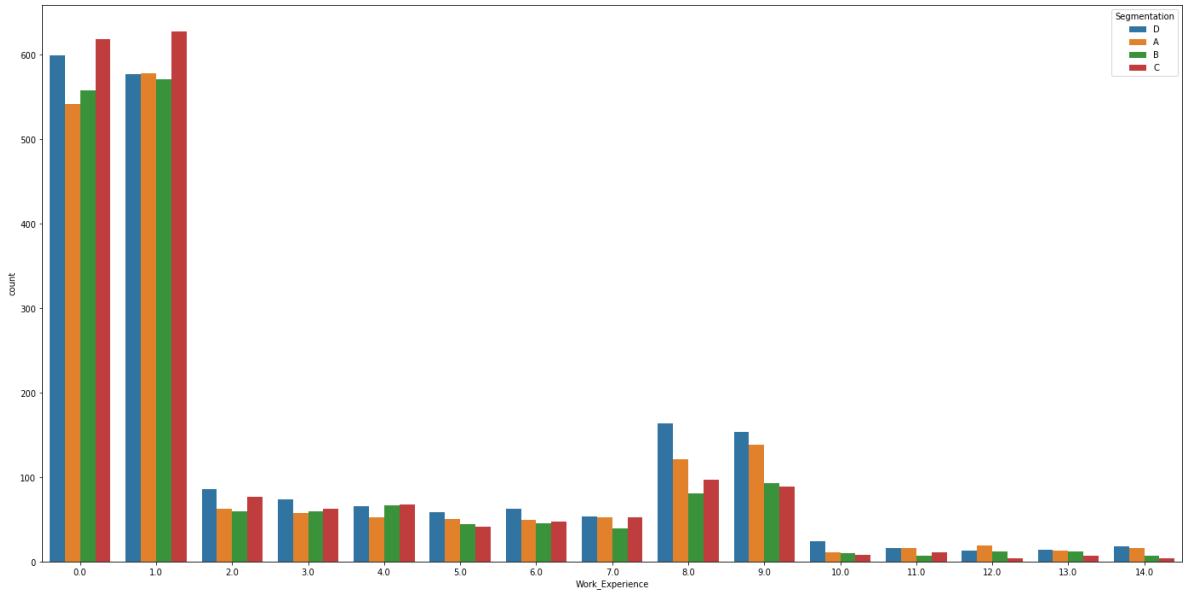


In [18]:

```
# Grouping our segments according to profession
sns.countplot('Work_Experience', hue = 'Segmentation', data = train)
```

Out[18]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x241357991c8>
```

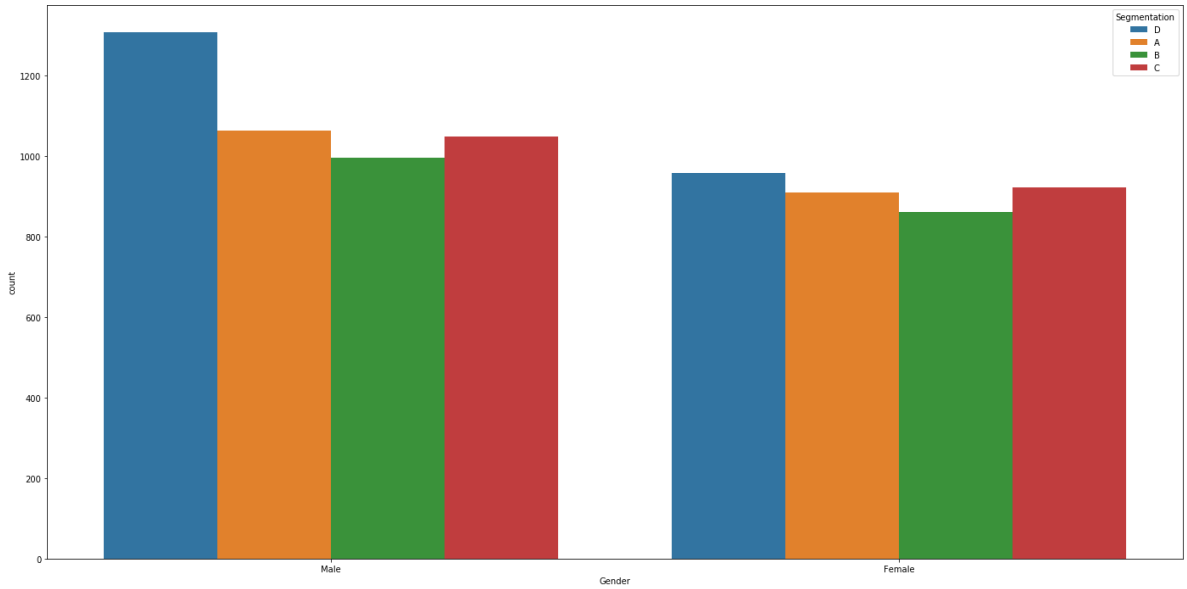


In [19]:

```
# Grouping our segments according to gender
sns.countplot('Gender', hue = 'Segmentation', data = train)
```

Out[19]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x24135c68fc8>
```



Visualizations doesn't show much information about what feature dominates a particular segmentation. So, we would need an algorithm like KNN or other deep learning models to get the insights.

Preprocessing

In [20]:

```
train['Family_Size'].value_counts()
```

Out[20]:

```
2.0    2390
3.0    1497
1.0    1453
4.0    1379
5.0     612
6.0     212
7.0      96
8.0      50
9.0      44
Name: Family_Size, dtype: int64
```

In [21]:

```
median_family_size = train['Family_Size'].median()
median_family_size
```

Out[21]:

```
3.0
```

In [22]:

```
# Replacing the missing values with the median value
train['Family_Size'].replace(np.nan, median_family_size, inplace = True)
train['Family_Size'].value_counts()
```

Out[22]:

```
2.0    2390
3.0    1832
1.0    1453
4.0    1379
5.0     612
6.0     212
7.0      96
8.0      50
9.0      44
Name: Family_Size, dtype: int64
```

In [23]:

```
median_Work_Experience = train['Work_Experience'].median()
median_Work_Experience
```

Out[23]:

```
1.0
```

In [24]:

```
# Replacing the missing values with the median value
train['Work_Experience'].replace(np.nan, median_Work_Experience, inplace = True)
train.isnull().sum()
```

Out[24]:

```
Gender               0
Ever_Married       140
Age                  0
Graduated           78
Profession         124
Work_Experience      0
Spending_Score       0
Family_Size          0
Segmentation         0
dtype: int64
```

In [25]:

```
# Replacing the missing profession values
train['Profession'].replace(np.nan, 'Not stated', inplace = True)
```

In [26]:

```
train['Profession'].unique()
```

Out[26]:

```
array(['Healthcare', 'Engineer', 'Lawyer', 'Entertainment', 'Artist',
       'Executive', 'Doctor', 'Homemaker', 'Marketing', 'Not stated'],
      dtype=object)
```

In [27]:

```
# To get the datatype for the features
train.dtypes
```

Out[27]:

```
Gender              object
Ever_Married        object
Age                  int64
Graduated           object
Profession          object
Work_Experience    float64
Spending_Score      object
Family_Size        float64
Segmentation        object
dtype: object
```

In [28]:

```
train.isnull().sum()
```

Out[28]:

```
Gender               0
Ever_Married       140
Age                  0
Graduated           78
Profession           0
Work_Experience      0
Spending_Score       0
Family_Size          0
Segmentation         0
dtype: int64
```

In [29]:

```
#train.dropna(inplace = True)
train.isnull().sum()
```

Out[29]:

```
Gender               0
Ever_Married       140
Age                  0
Graduated           78
Profession           0
Work_Experience      0
Spending_Score       0
Family_Size          0
Segmentation         0
dtype: int64
```

In [30]:

```
# Casting the datatypes of selected features to strings
train['Gender'] = train['Gender'].astype('str')
train['Ever_Married'] = train['Ever_Married'].astype('str')
train['Graduated'] = train['Graduated'].astype('str')
```

In [31]:

```
# Encoding our variable to enable our model work effectively
enc = LabelEncoder()
train['gender'] = enc.fit_transform(train['Gender'])
train['ever_married'] = enc.fit_transform(train['Ever_Married'])
train['graduated'] = enc.fit_transform(train['Graduated'])
train.head()
```

Out[31]:

	Gender	Ever_Married	Age	Graduated	Profession	Work_Experience	Spending_Score	Fa
0	Male	No	22	No	Healthcare	1.0	Low	
1	Female	Yes	38	Yes	Engineer	1.0	Average	
2	Female	Yes	67	Yes	Engineer	1.0	Low	
3	Male	Yes	67	Yes	Lawyer	0.0	High	
4	Female	Yes	40	Yes	Entertainment	1.0	High	

In [32]:

```
# Dropping the encoded features
df = train.drop(['Gender', 'Ever_Married', 'Graduated'], axis = 1)
df.head()
```

Out[32]:

	Age	Profession	Work_Experience	Spending_Score	Family_Size	Segmentation	gender	e
0	22	Healthcare	1.0	Low	4.0	D	1	
1	38	Engineer	1.0	Average	3.0	A	0	
2	67	Engineer	1.0	Low	1.0	B	0	
3	67	Lawyer	0.0	High	2.0	B	1	
4	40	Entertainment	1.0	High	6.0	A	0	

In [33]:

```
#df = df.drop(['Profession'], axis = 1)
```

In [34]:

```
# Converting the strings in Spending column to integer
mapping = {'Low': 1, 'Average': 2, 'High': 3}
df['Spending_Score'] = df['Spending_Score'].map(mapping)
df.head()
```

Out[34]:

	Age	Profession	Work_Experience	Spending_Score	Family_Size	Segmentation	gender	e
0	22	Healthcare	1.0	1	4.0	D	1	
1	38	Engineer	1.0	2	3.0	A	0	
2	67	Engineer	1.0	1	1.0	B	0	
3	67	Lawyer	0.0	3	2.0	B	1	
4	40	Entertainment	1.0	3	6.0	A	0	

In [35]:

```
#Creating dummy variables from profession column
small_df = pd.get_dummies(df['Profession'])
small_df
```

Out[35]:

	Artist	Doctor	Engineer	Entertainment	Executive	Healthcare	Homemaker	Lawyer	Mar
0	0	0	0	0	0	1	0	0	
1	0	0	1	0	0	0	0	0	
2	0	0	1	0	0	0	0	0	
3	0	0	0	0	0	0	0	1	
4	0	0	0	1	0	0	0	0	
...	...	...	...	...	...	...	...	...	
8063	0	0	0	0	0	0	0	0	
8064	0	0	0	0	1	0	0	0	
8065	0	0	0	0	0	1	0	0	
8066	0	0	0	0	0	1	0	0	
8067	0	0	0	0	1	0	0	0	

8068 rows × 10 columns

In [36]:

```
# Adding the dummy values to the main dataframe
df = pd.concat([df, small_df], axis = 1)
df
```

Out[36]:

	Age	Profession	Work_Experience	Spending_Score	Family_Size	Segmentation	gende
0	22	Healthcare	1.0	1	4.0	D	1
1	38	Engineer	1.0	2	3.0	A	0
2	67	Engineer	1.0	1	1.0	B	0
3	67	Lawyer	0.0	3	2.0	B	1
4	40	Entertainment	1.0	3	6.0	A	0
...	...	...	...	...	...	...	..
8063	22	Not stated	0.0	1	7.0	D	1
8064	35	Executive	3.0	1	4.0	D	1
8065	33	Healthcare	1.0	1	1.0	D	0
8066	27	Healthcare	1.0	1	4.0	B	0
8067	37	Executive	0.0	2	3.0	B	1

8068 rows × 19 columns

In [37]:

```
#Dropping profession column after adding the dummy values
df.drop(['Profession'], axis = 1, inplace = True)
```

In [38]:

```
df.head()
```

Out[38]:

	Age	Work_Experience	Spending_Score	Family_Size	Segmentation	gender	ever_married	g
0	22	1.0	1	4.0	D	1	0	
1	38	1.0	2	3.0	A	0	1	
2	67	1.0	1	1.0	B	0	1	
3	67	0.0	3	2.0	B	1	1	
4	40	1.0	3	6.0	A	0	1	

In [39]:

```
df.shape
```

Out[39]:

```
(8068, 18)
```

In [40]:

```
# X, y = train.drop('Segmentation', axis = 1), train['Segmentation']
# X.head()
```

In [41]:

```
# Assigning our target and features
X, y = df.drop('Segmentation', axis = 1), df['Segmentation']
X.head()
```

Out[41]:

	Age	Work_Experience	Spending_Score	Family_Size	gender	ever_married	graduated	Artis
0	22	1.0	1	4.0	1	0	0	
1	38	1.0	2	3.0	0	1	1	
2	67	1.0	1	1.0	0	1	1	
3	67	0.0	3	2.0	1	1	1	
4	40	1.0	3	6.0	0	1	1	

In [42]:

```
#X = df[['Age', 'Work_Experience']]
```

In [43]:

```
y
```

Out[43]:

```
0       D
1       A
2       B
3       B
4       A
       ..
8063    D
8064    D
8065    D
8066    B
8067    B
Name: Segmentation, Length: 8068, dtype: object
```

In [44]:

```
# scaler = MinMaxScaler()
# scaled_df = scaler.fit_transform(X)
# scaled_df[:10]
```

In [45]:

```
# Scaling our features so that the large values don't have much extra effect on the model t
scaler = MinMaxScaler()
X['scaled_age'] = scaler.fit_transform(X['Age'].values.reshape(-1,1))
X['scaled_Work_Experience'] = scaler.fit_transform(X['Work_Experience'].values.reshape(-1,1
X['scaled_Family_size'] = scaler.fit_transform(X['Family_Size'].values.reshape(-1,1))
X.head()
```

Out[45]:

	Age	Work_Experience	Spending_Score	Family_Size	gender	ever_married	graduated	Artis
0	22	1.0	1	4.0	1	0	0	
1	38	1.0	2	3.0	0	1	1	
2	67	1.0	1	1.0	0	1	1	
3	67	0.0	3	2.0	1	1	1	
4	40	1.0	3	6.0	0	1	1	

In [46]:

```
X['scaled_age'].max()
X['Age'].max()
```

Out[46]:

89

In [47]:

```
#Dropping the original, unscaled features
X.drop(['Age', 'Work_Experience','Family_Size'], axis = 1, inplace = True)
```

In [48]:

```
X.head()
```

Out[48]:

	Spending_Score	gender	ever_married	graduated	Artist	Doctor	Engineer	Entertainment
0	1	1	0	0	0	0	0	0
1	2	0	1	1	0	0	1	0
2	1	0	1	1	0	0	1	0
3	3	1	1	1	0	0	0	0
4	3	0	1	1	0	0	0	1

In [49]:

```
#Splitting our dataset in preparation for model application
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y)
```

In [50]:

```
#Instantiating and fitting our model
knn = KNeighborsClassifier(n_neighbors = 44, leaf_size = 15)
knn.fit(X_train, y_train)
```

Out[50]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=15, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=44, p=2,
                     weights='uniform')
```

In [51]:

```
# Predicting our model
y_pred = knn.predict(X_test)
```

In [52]:

```
# Checking our training score
print(accuracy_score(y_train, knn.predict(X_train)))
```

```
0.5374318294496777
```

In [53]:

```
# checking our testing score
print(accuracy_score(y_test, y_pred))
```

```
0.5101636093207734
```

In [54]:

```
y_pred[:10]
```

Out[54]:

```
array(['B', 'A', 'B', 'D', 'B', 'B', 'B', 'C', 'A', 'B'], dtype=object)
```

In [55]:

```
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
#print(roc_auc_score(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           A       0.43      0.44      0.44       493
           B       0.36      0.36      0.36       465
           C       0.56      0.55      0.56       492
           D       0.65      0.66      0.66       567

    accuracy                           0.51      2017
   macro avg       0.50      0.50      0.50      2017
weighted avg       0.51      0.51      0.51      2017

[[217 141  51  84]
 [103 168 142  52]
 [ 57 104 270  61]
 [127  51  15 374]]
```

In [56]:

```
for i in range(1,6):
    print(2 * i)
```

```
2
4
6
8
10
```

In [57]:

```
# List Hyperparameters that we want to tune
leaf_size = 2 * list(range(1,10))
n_neighbors = 4 * list(range(1,15))
p = [1,2]

#Convert to dictionary
hyperparameters = dict(leaf_size = leaf_size, n_neighbors = n_neighbors, p = p)

#Creating a new KNN model
knn2 = KNeighborsClassifier()

#Use gridsearch
clf = GridSearchCV(knn2, hyperparameters, cv = 10)

#fit the model
best_model = clf.fit(X_train, y_train)
```

In [58]:

```
print('Best leaf_size: {}'.format(best_model.best_estimator_.get_params()['leaf_size']))
print('Best p: {}'.format(best_model.best_estimator_.get_params()['p']))
print('Best n_neighbors: {}'.format(best_model.best_estimator_.get_params()['n_neighbors']))
```

```
Best leaf_size: 6
Best p: 2
Best n_neighbors: 14
```

In [82]:

```
knn3 = KNeighborsClassifier(p = 2, leaf_size = 12, n_neighbors = 56)
knn3.fit(X_train, y_train)
y_pred = knn3.predict(X_test)
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           A       0.44      0.43      0.44       493
           B       0.36      0.37      0.36       465
           C       0.56      0.55      0.56       492
           D       0.66      0.66      0.66       567

    accuracy                           0.51      2017
   macro avg       0.50      0.50      0.50      2017
weighted avg       0.51      0.51      0.51      2017

[[214 140  54  85]
 [105 171 141  48]
 [ 46 114 271  61]
 [124  54  17 372]]
```

In [83]:

```
# checking our testing score
print(accuracy_score(y_test, y_pred))
```

```
0.5096678235002479
```

In [61]:

```
print(best_model.get_params())
```

```
{'cv': 10, 'error_score': nan, 'estimator__algorithm': 'auto', 'estimator__l
eaf_size': 30, 'estimator__metric': 'minkowski', 'estimator__metric_params':
None, 'estimator__n_jobs': None, 'estimator__n_neighbors': 5, 'estimator__
p': 2, 'estimator__weights': 'uniform', 'estimator': KNeighborsClassifier(al
gorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform'), 'iid': 'deprecated', 'n_jobs': Non
e, 'param_grid': {'leaf_size': [1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6,
7, 8, 9], 'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1,
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1
1, 12, 13, 14, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14], 'p': [1, 2]},
'pre_dispatch': '2*n_jobs', 'refit': True, 'return_train_score': False, 'sco
ring': None, 'verbose': 0}
```

Segmenting our test dataset using the best model

In [85]:

```
test = pd.read_csv('test.csv')
test.head()
```

Out[85]:

	ID	Gender	Ever_Married	Age	Graduated	Profession	Work_Experience	Spending_Sc
0	458989	Female	Yes	36	Yes	Engineer	0.0	L
1	458994	Male	Yes	37	Yes	Healthcare	8.0	Avera
2	458996	Female	Yes	69	No	NaN	0.0	L
3	459000	Male	Yes	59	No	Executive	11.0	H
4	459001	Female	No	19	No	Marketing	NaN	L

In [86]:

```
test = test.drop(['ID', 'Var_1'], axis = 1)
```

In [87]:

```
test.head()
```

Out[87]:

	Gender	Ever_Married	Age	Graduated	Profession	Work_Experience	Spending_Score	Fam
0	Female	Yes	36	Yes	Engineer	0.0	Low	
1	Male	Yes	37	Yes	Healthcare	8.0	Average	
2	Female	Yes	69	No	NaN	0.0	Low	
3	Male	Yes	59	No	Executive	11.0	High	
4	Female	No	19	No	Marketing	NaN	Low	

In [88]:

```
median_family_size = test['Family_Size'].median()

# Replacing the missing values with the median value
test['Family_Size'].replace(np.nan, median_family_size, inplace = True)
test['Family_Size'].value_counts()
```

Out[88]:

```
2.0    881
1.0    512
3.0    455
4.0    444
5.0    200
6.0     78
7.0     26
9.0     16
8.0     15
Name: Family_Size, dtype: int64
```

In [89]:

```
median_Work_Experience = test['Work_Experience'].median()
median_Work_Experience
# Replacing the missing values with the median value
test['Work_Experience'].replace(np.nan, median_Work_Experience, inplace = True)
test.isnull().sum()
```

Out[89]:

```
Gender              0
Ever_Married       50
Age                 0
Graduated          24
Profession         38
Work_Experience     0
Spending_Score      0
Family_Size         0
dtype: int64
```

In [90]:

```
# Replacing the missing profession values
test['Profession'].replace(np.nan, 'Not stated', inplace = True)
```

In [91]:

```
test.dropna(inplace = True)
test.isnull().sum()
```

Out[91]:

```
Gender             0
Ever_Married       0
Age                0
Graduated          0
Profession         0
Work_Experience    0
Spending_Score     0
Family_Size        0
dtype: int64
```

In [92]:

```
# Casting the datatypes of selected features to strings
test['Gender'] = test['Gender'].astype('str')
test['Ever_Married'] = test['Ever_Married'].astype('str')
test['Graduated'] = test['Graduated'].astype('str')
```

In [93]:

```
# Encoding our variable to enable our model work effectively
enc = LabelEncoder()
test['gender'] = enc.fit_transform(test['Gender'])
test['ever_married'] = enc.fit_transform(test['Ever_Married'])
test['graduated'] = enc.fit_transform(test['Graduated'])
test.head()
```

Out[93]:

	Gender	Ever_Married	Age	Graduated	Profession	Work_Experience	Spending_Score	Fam
0	Female	Yes	36	Yes	Engineer	0.0	Low	
1	Male	Yes	37	Yes	Healthcare	8.0	Average	
2	Female	Yes	69	No	Not stated	0.0	Low	
3	Male	Yes	59	No	Executive	11.0	High	
4	Female	No	19	No	Marketing	1.0	Low	

In [ ]:

```

```

In [94]:

```
test.drop(['Gender', 'Ever_Married', 'Graduated'], axis = 1, inplace = True)
```

In [95]:

```
mapping = {'Low': 1, 'Average': 2, 'High': 3}
test['Spending_Score'] = test['Spending_Score'].map(mapping)
test.head()
```

Out[95]:

	Age	Profession	Work_Experience	Spending_Score	Family_Size	gender	ever_married	gra
0	36	Engineer	0.0	1	1.0	0	1	
1	37	Healthcare	8.0	2	4.0	1	1	
2	69	Not stated	0.0	1	1.0	0	1	
3	59	Executive	11.0	3	2.0	1	1	
4	19	Marketing	1.0	1	4.0	0	0	

In [96]:

```
#Creating dummy variables from profession column
small_test = pd.get_dummies(test['Profession'])
small_test
```

Out[96]:

	Artist	Doctor	Engineer	Entertainment	Executive	Healthcare	Homemaker	Lawyer	Mar
0	0	0	1	0	0	0	0	0	
1	0	0	0	0	0	1	0	0	
2	0	0	0	0	0	0	0	0	
3	0	0	0	0	1	0	0	0	
4	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	
2622	0	0	0	0	0	1	0	0	
2623	0	1	0	0	0	0	0	0	
2624	0	0	0	1	0	0	0	0	
2625	0	0	0	0	1	0	0	0	
2626	0	0	0	0	0	1	0	0	

2554 rows × 10 columns

In [97]:

```
# Adding the dummy values to the main dataframe
test = pd.concat([test, small_test], axis = 1)
test
```

Out[97]:

	Age	Profession	Work_Experience	Spending_Score	Family_Size	gender	ever_married
0	36	Engineer	0.0	1	1.0	0	1
1	37	Healthcare	8.0	2	4.0	1	1
2	69	Not stated	0.0	1	1.0	0	1
3	59	Executive	11.0	3	2.0	1	1
4	19	Marketing	1.0	1	4.0	0	0
...	...	...	...	...	...	...	...
2622	29	Healthcare	9.0	1	4.0	1	0
2623	35	Doctor	1.0	1	1.0	0	0
2624	53	Entertainment	1.0	1	2.0	0	0
2625	47	Executive	1.0	3	5.0	1	1
2626	43	Healthcare	9.0	1	3.0	0	0

2554 rows × 18 columns

In [98]:

```
test.drop('Profession', axis = 1, inplace = True)
```

In [99]:

```
test.head()
```

Out[99]:

	Age	Work_Experience	Spending_Score	Family_Size	gender	ever_married	graduated	Artis
0	36	0.0	1	1.0	0	1	1	
1	37	8.0	2	4.0	1	1	1	
2	69	0.0	1	1.0	0	1	0	
3	59	11.0	3	2.0	1	1	0	
4	19	1.0	1	4.0	0	0	0	

In [100]:

```
# Scaling our features so that the large values don't have much extra effect on the model t
scaler = MinMaxScaler()
test['scaled_age'] = scaler.fit_transform(test['Age'].values.reshape(-1,1))
test['scaled_Work_Experience'] = scaler.fit_transform(test['Work_Experience'].values.reshap
test['scaled_Family_size'] = scaler.fit_transform(test['Family_Size'].values.reshape(-1,1))
test.head()
```

Out[100]:

	Age	Work_Experience	Spending_Score	Family_Size	gender	ever_married	graduated	Artis
0	36	0.0	1	1.0	0	1	1	
1	37	8.0	2	4.0	1	1	1	
2	69	0.0	1	1.0	0	1	0	
3	59	11.0	3	2.0	1	1	0	
4	19	1.0	1	4.0	0	0	0	

In [101]:

```
#Dropping the original, unscaled features
test.drop(['Age', 'Work_Experience','Family_Size'], axis = 1, inplace = True)
```

In [102]:

```
segments = knn3.predict(test)
segments[:10]

test['Segmentation'] = segments

test.head()
```

Out[102]:

	Spending_Score	gender	ever_married	graduated	Artist	Doctor	Engineer	Entertainment
0	1	0	1	1	0	0	1	0
1	2	1	1	1	0	0	0	0
2	1	0	1	0	0	0	0	0
3	3	1	1	0	0	0	0	0
4	1	0	0	0	0	0	0	0

In [103]:

```
test.to_csv('segmented_test.csv')
```

Conclusion

- After trying several other combinations of features, the combination that resulted in the highest testing

After trying several other combinations of features, the combination that resulted in the highest testing score was one that used all the relevant features with scaling and encoding where appropriate. The scores we got are still low.

- Visualizations didn't show any distinct segment where a feature dominates
- Training score is usually low when we dont have enough entries in our dataset but that isn't the case here. It's likely that the segmentation was done randomly or the dataset have missing information.

Type *Markdown* and LaTeX: α^2