Capstone Report

Customer Segmentation and Acquisition Optimization for Arvato Financial Solutions

Shilton Jonatan Salindeho

I. Definition

Project Overview

The project simulates a real-world problem where one of Arvato's client, a mail-order company, who is interested to acquire new clients more efficiently.

Techniques employed in order to tackle this problem are customer segmentation in order to identify parts of the population that best describe the core customer base, combining two datasets of general population and the company's customer base; as well as predictive analytics in order to predict which individuals are most likely to convert into becoming the company's customer from a third dataset with demographic information.

This project is a requirement for the completion of the 'Machine Learning Engineer' Nanodegree at Udacity in collaboration with Temasek Polytechnic as part of its knowITgetIT program. It was made possible by the combined efforts of Udacity and Arvato Financial Services, who kindly gave us (restricted) access to their data – all datasets used in this project are their sole private property (refer to Terms & Conditions) and are inclusively used for the development of this project alone (single-use only).

Problem Statement

The problem statement of this project is as follows:

"How can the client (the mail-order company) convert new customers more efficiently, provided overall German demographics data, customer base data, and potential new customers data?"

More specifically, we are trying to predict whether someone is likely to become a customer of our client provided his/her demographic data. This is possible to quantify and measure since from segmenting our initial data (overall demographics and existing customer base) we will be able to construct a model that will be able to calculate the probabilities based on how similar the potential new customers are to the existing segments/clusters.

Datasets and Inputs

The project makes use of four datasets:

- Udacity_AZDIAS_052018.csv: Demographics data for the general population of Germany; 891 211 persons (rows) x 366 features (columns)
- Udacity_CUSTOMERS_052018.csv: Demographics data for customers of a mail-order company; 191 652 persons (rows) x 369 features (columns)
- Udacity_MAILOUT_052018_TRAIN.csv: Demographics data for individuals who were targets of a marketing campaign; 42 982 persons (rows) x 367 (columns)
- Udacity_MAILOUT_052018_TEST.csv: Demographics data for individuals who were targets of a marketing campaign; 42 833 persons (rows) x 366 (columns)

Benchmark Model

Our benchmark model for the second stage of the project (supervised learning, classification) will be the Logistic Regression classifier since this is the standard model with 1 as "converted into new customer" score and 0 as the "not converted into new customer" score.

Evaluation Metrics

The possible metric choices for our evaluation are: F1 score, precision, recall, and area under the receiver operating curve (ROC) otherwise known as AUC.

We will be using the AUC to evaluate performance of different models because it is one of the best options for the imbalanced data. AUC can be interpreted as the probability that the model ranks a random positive example more highly than a random negative example. In addition, the Kaggle Competition also uses AUC as the evaluation metric.

II. Analysis

Data Exploration

In data exploration, looking at the datasets at hand along with the provided metadata files is helpful for us to better understand the meaning of features, the range of unknown and missing values, and the range of values that features can have.

Some top level Pandas methods to get an initial sense of how the data is structured:

AZDIAS Dataset

```
In [3]: azdias.head()
Out[3]:
```

	LNR	AGER_TYP	AKT_DAT_KL	ALTER_HH	ALTER_KIND1	ALTER_KIND2	ALTER_KIND3	ALTER_KIND4	ALTERSKATEGORIE_FEIN	ANZ_HAUSHALTE_AKTIV
0	910215	-1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	910220	-1	9.0	0.0	NaN	NaN	NaN	NaN	21.0	11.0
2	910225	-1	9.0	17.0	NaN	NaN	NaN	NaN	17.0	10.0
3	910226	2	1.0	13.0	NaN	NaN	NaN	NaN	13.0	1.0
4	910241	-1	1.0	20.0	NaN	NaN	NaN	NaN	14.0	3.0

5 rows × 366 columns

```
In [4]: azdias.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 891221 entries, 0 to 891220
Columns: 366 entries, LNR to ALTERSKATEGORIE_GROB
dtypes: float64(267), int64(93), object(6)
memory usage: 2.4+ GB

In [5]: azdias.shape
Out[5]: (891221, 366)
```

CUSTOMERS Dataset

In [6]:
```
customers.head()
```

Out[6]:

	LNR	AGER_TYP	AKT_DAT_KL	ALTER_HH	ALTER_KIND1	ALTER_KIND2	ALTER_KIND3	ALTER_KIND4	ALTERSKATEGORIE_FEIN	ANZ_HAUSHALTE_AKTIV
0	9626	2	1.0	10.0	NaN	NaN	NaN	NaN	10.0	1.0
1	9628	-1	9.0	11.0	NaN	NaN	NaN	NaN	NaN	NaN
2	143872	-1	1.0	6.0	NaN	NaN	NaN	NaN	0.0	1.0
3	143873	1	1.0	8.0	NaN	NaN	NaN	NaN	8.0	0.0
4	143874	-1	1.0	20.0	NaN	NaN	NaN	NaN	14.0	7.0

5 rows × 369 columns

In [7]:
```
customers.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 191652 entries, 0 to 191651
Columns: 369 entries, LNR to ALTERSKATEGORIE_GROB
dtypes: float64(267), int64(94), object(8)
memory usage: 541.0+ MB
```

In [8]:
```
customers.shape
```

Out[8]: (191652, 369)

As seen from the code screenshot, the AZDIAS dataset contains 891k rows and 366 features. This dataset is the demographic data for the general population of Germany.

On the other hand, we can also notice that the customers dataset contains 191k and 369 features. This dataset is the demographic data for the customers of Arvato.

In [9]:
```
attributes = pd.read_csv('attributes metadata v2.csv')
```

In [10]:
```
attributes.head()
```

Out[10]:

	Attributes	Variable Type	Missing Values
0	AGER_TYP	categorical	[-1,0]
1	ALTERSKATEGORIE_FEIN	ordinal	[-1,0,9]
2	ALTERSKATEGORIE_GROB	ordinal	[-1,0,9]
3	ALTER_HH	ordinal	[0]
4	ANREDE_KZ	categorical	[-1,0]

In [11]:
```
attributes.shape
```

Out[11]: (334, 3)

We also did some exploration on the provided .xlsx files that gave us more information regarding the features and we then created a separate metadata file attributes.csv which contains the attribute name, variable type (e.g. categorical, ordinal), and the list of possible missing values that the feature may have - refer to the code snippet above.

Data Preprocessing

```
In [14]: to_drop_nonexistent = list(set(azdias) - set(features_in_att))
         to_drop_nonexistent_c = list(set(customers) - set(features_in_att))

In [15]: azdias.drop(labels=to_drop_nonexistent, axis=1, inplace=True)
         customers.drop(labels=to_drop_nonexistent_c, axis=1, inplace=True)

In [16]: len(to_drop_nonexistent)

Out[16]: 32

In [17]: len(to_drop_nonexistent_c)

Out[17]: 35

In [18]: azdias.shape

Out[18]: (891221, 334)

In [19]: customers.shape

Out[19]: (191652, 334)
```

We then tried to see whether we can remove features in both AZDIAS and customers dataset that are absent from the attributes.csv file. As seen from the code snippet, we eventually ended up with 334 features for both datasets.

```
#Convert to float - Azdias
for column in azdias.columns:
    if azdias[column].dtype == np.int64:
        azdias[column] = azdias[column].astype(np.float64)
```

```
#Convert to float - Customers
for column in customers.columns:
    if customers[column].dtype == np.int64:
        customers[column] = customers[column].astype(np.float64)
```

Since there may be features with numbers in them that are considered float by the metadata but are read as integers by Python, we will need to convert them to float.

```
missing_value_series = pd.Series(attributes['Missing Values'].values, index=attributes['Attributes'])
```

```
missing_value_series
```

```
Attributes
AGER_TYP                  [-1,0]
ALTERSKATEGORIE_FEIN    [-1,0.9]
ALTERSKATEGORIE_GROB    [-1,0,9]
ALTER_HH                     [0]
ANREDE_KZ                 [-1,0]
                           ...
VERS_TYP                    [-1]
WOHNDAUER_2008            [-1,0]
WOHNLAGE                  [-1,0]
W_KEIT_KIND_HH            [-1,0]
ZABEOTYP                  [-1,9]
Length: 334, dtype: object
```

```
#Label missing value - Azdias
for column in azdias.columns:

    isin = ast.literal_eval(missing_value_series[column])

    azdias[column] = azdias[column].mask(azdias[column].isin(isin), other=np.nan)
```

```
#Label missing value - Customers
for column in customers.columns:

    isin = ast.literal_eval(missing_value_series[column])

    customers[column] = customers[column].mask(customers[column].isin(isin), other=np.nan)
```

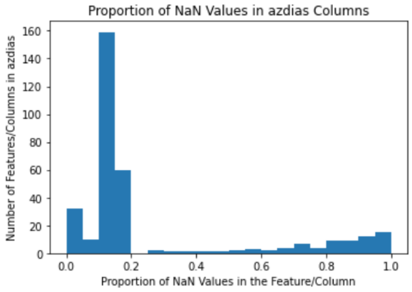
```
missing_perct_column = azdias.isnull().mean(axis=0)
missing_perct_column
```

Also, the metadata gives us the possible values in each features that we can consider as missing - we marked them as NaN instead of their original value.

```
plt.hist(missing_perct_column, bins=20);

plt.xlabel('Proportion of NaN Values in the Feature/Column')
plt.ylabel('Number of Features/Columns in azdias')
plt.title('Proportion of NaN Values in azdias Columns')
```

```
Text(0.5, 1.0, 'Proportion of NaN Values in azdias Columns')
```




```
to_drop_missing = missing_perct_column[missing_perct_column > 0.2].index
azdias.drop(labels=to_drop_missing, axis=1, inplace=True)
customers.drop(labels=to_drop_missing, axis=1, inplace=True)
```

```
azdias.shape
```

```
(891221, 261)
```

```
customers.shape
```

```
(191652, 261)
```

After plotting the proportion of NaN values in each AZDIAS columns, we noticed that the majority of features do not have that many NaN entries; most have 20% or less NaN values. Due to this, we decided that we will remove columns which have more than 20% NaN values for both AZDIAS and customers columns.

```
In [31]: missing_perct_row = azdias.isnull().mean(axis=1)
         missing_perct_row

Out[31]: 0         0.862069
         1         0.000000
         2         0.000000
         3         0.030651
         4         0.000000
                     ...
         891216    0.011494
         891217    0.015326
         891218    0.022989
         891219    0.000000
         891220    0.000000
         Length: 891221, dtype: float64

In [32]: azdias = azdias[missing_perct_row < 0.2]
         customers = customers[missing_perct_row < 0.2]

In [33]: azdias.shape

Out[33]: (737288, 261)

In [34]: customers.shape

Out[34]: (159036, 261)
```

Similarly we can also investigate the NaN values by row, and for simplicity and consistency sake, we will also remove rows which contain 20% or more NaN values for both AZDIAS and customers dataset.

Re-encoding of Features

```
In [37]: #categorical features from list of attributes, not dataset
         categorical_features_attributes = attributes['Attributes'].loc[attributes['Variable Type'] == 'categorical']

         categorical_features_attributes

Out[37]: 0              AGER_TYP
         4             ANREDE_KZ
         13      CAMEO_DEUG_2015
         14       CAMEO_DEU_2015
         16         CJT_GESAMTTYP
         46         D19_KONSUMTYP
         83              DSL_FLAG
         86             FINANZTYP
         93           GEBAEUDETYP
         96       GFK_URLAUBERTYP
         97      GREEN_AVANTGARDE
         99         HH_DELTA_FLAG
         284         KK_KUNDENTYP
         286          KONSUMZELLE
         287      LP_FAMILIE_FEIN
         288      LP_FAMILIE_GROB
         291       LP_STATUS_FEIN
         292       LP_STATUS_GROB
         296     NATIONALITAET_KZ
         299          OST_WEST_KZ
         325          SHOPPER_TYP
         326              SOHO_KZ
         327             TITEL_KZ
         328     UNGLEICHENN_FLAG
         329             VERS_TYP
         333            ZABEOTYP
         Name: Attributes, dtype: object
```

Then, we will re-encode categorical and mixed features into numerical features for processing purposes. Re-encoding steps will be done separately for categorical and mixed.

1. Categorical to Numerical

```
In [43]: cat_binary_numerical = []
         cat_binary_nonnumerical = []
         cat_multilevel = []

         for attribute in categorical_features_azdias:
             dtype = azdias[attribute].dtype
             count = len(azdias[attribute].value_counts())

             # if multi-level categorical feature
             if count > 2:
                 cat_multilevel.append(attribute)
             else:
                 if dtype == 'object':
                     cat_binary_nonnumerical.append(attribute)
                 else:
                     cat_binary_numerical.append(attribute)

In [44]: cat_binary_numerical

Out[44]: ['ANREDE_KZ',
          'DSL_FLAG',
          'GREEN_AVANTGARDE',
          'HH_DELTA_FLAG',
          'KONSUMZELLE',
          'SOHO_KZ',
          'UNGLEICHENN_FLAG',
          'VERS_TYP']

In [45]: cat_binary_nonnumerical

Out[45]: ['OST_WEST_KZ']

In [46]: cat_multilevel

Out[46]: ['CAMEO_DEUG_2015',
          'CAMEO_DEU_2015',
          'CJT_GESAMTTYP',
          'FINANZTYP',
          'GEBAEUDETYP',
          'GFK_URLAUBERTYP',
          'LP_FAMILIE_FEIN',
          'LP_FAMILIE_GROB',
          'LP_STATUS_FEIN',
          'LP_STATUS_GROB',
          'NATIONALITAET_KZ',
          'SHOPPER_TYP',
          'ZABEOTYP']
```

```
In [51]: #Re-encoding Binary Non-Numerical

         azdias['OST_WEST_KZ'] = azdias['OST_WEST_KZ'].map({'W': 1, 'O': 2})
         customers['OST_WEST_KZ'] = customers['OST_WEST_KZ'].map({'W': 1, 'O': 2})

         #Re-encoding MultiLevel

         list_columns_to_add = []
         list_columns_to_add_c = []

         #AZDIAS
         for column in cat_multilevel:

             # delete features with 10 or more levels - only select those with less than 10
             if len(azdias[column].value_counts()) < 10:
                 list_columns_to_add.append(pd.get_dummies(azdias[column], prefix=column))


         # drop the original
         azdias.drop(cat_multilevel, axis=1, inplace=True)

         list_columns_to_add.append(azdias)

         # add the re-encoded
         azdias = pd.concat(list_columns_to_add, axis=1)
```

For categorical, we split the columns into three types: multilevel (those with more than two possible values), binary non-numerical (two possible values, data type is object), and binary numerical (two possible values, data type is numerical). All these are then treated separately - we add multiple columns for the multilevel ones to handle each possible values (using Pandas' get_dummies method), while we change the values of the binary non-numerical ones into numerical. Once done, the old multi-level variables are then dropped, and the new ones added.

```
In [53]: # Check
         azdias.head()
```

Out[53]:

	CJT_GESAMTTYP_1.0	CJT_GESAMTTYP_2.0	CJT_GESAMTTYP_3.0	CJT_GESAMTTYP_4.0	CJT_GESAMTTYP_5.0	CJT_GESAMTTYP_6.0	FINANZTYP_1.0	FINANZTYP_2.0	F
1	0	0	0	0	1	0	1	0	
2	0	0	1	0	0	0	1	0	
3	0	1	0	0	0	0	0	0	
4	0	0	0	0	1	0	0	0	
5	0	1	0	0	0	0	0	1	

5 rows × 290 columns

```
In [54]: azdias.shape
Out[54]: (737288, 290)

In [55]: customers.head()
```

Out[55]:

	CJT_GESAMTTYP_1.0	CJT_GESAMTTYP_2.0	CJT_GESAMTTYP_3.0	CJT_GESAMTTYP_4.0	CJT_GESAMTTYP_5.0	CJT_GESAMTTYP_6.0	FINANZTYP_1.0	FINANZTYP_2.0	F
1	0	0	0	0	0	0	0	1	
2	0	1	0	0	0	0	0	1	
3	0	1	0	0	0	0	0	0	
4	0	0	0	0	0	1	0	1	
5	0	0	0	1	0	0	0	0	

5 rows × 289 columns

```
In [56]: customers.shape
Out[56]: (159036, 289)
```

We note that the customers dataset has one column short (`GEBAEUDETYP_5.0`) due to the value `GEBAEUDETYP` =5 not being present in the customer dataset.

Notice that the customer dataset has one column short compared to the AZDIAS dataset (GEBAEUDETYP_5.0) due to the lack of rows with GEBAEUDETYP value = 5 in customers.

```
In [58]: # Function to reencode 'PRAEGENDE_JUGENDJAHRE'
         def encode_pj(value):
             newvalue = 0
             if value in [2, 4, 6, 7, 9, 11, 13, 15]:
                 newvalue = 1
             return newvalue

In [59]: azdias['PRAEGENDE_JUGENDJAHRE'] = azdias['PRAEGENDE_JUGENDJAHRE'].apply(lambda x: encode_pj(x))
         customers['PRAEGENDE_JUGENDJAHRE'] = customers['PRAEGENDE_JUGENDJAHRE'].apply(lambda x: encode_pj(x))

In [60]: azdias.drop(['LP_LEBENSPHASE_FEIN', 'LP_LEBENSPHASE_GROB'], axis=1, inplace=True)
         customers.drop(['LP_LEBENSPHASE_FEIN', 'LP_LEBENSPHASE_GROB'], axis=1, inplace=True)

In [61]: azdias['WEALTH_LEVEL'] = azdias['CAMEO_INTL_2015'].apply(lambda x: np.floor(pd.to_numeric(x)/10))
         azdias['STATUS'] = azdias['CAMEO_INTL_2015'].apply(lambda x: pd.to_numeric(x)%10)

         customers['WEALTH_LEVEL'] = customers['CAMEO_INTL_2015'].apply(lambda x: np.floor(pd.to_numeric(x)/10))
         customers['STATUS'] = customers['CAMEO_INTL_2015'].apply(lambda x: pd.to_numeric(x)%10)

         # drop CAMEO_INTL_2015
         azdias.drop('CAMEO_INTL_2015', axis=1, inplace=True)
         customers.drop('CAMEO_INTL_2015', axis=1, inplace=True)

In [62]: azdias.shape
Out[62]: (737288, 289)

In [63]: customers.shape
Out[63]: (159036, 288)
```

For the mixed features, we decided to drop two columns (LP_LEBENSPHASE_FEIN and LP_LEBENSPHASE_GROB) due to them taking numerous possible values and thus making them quite complex compared to the other features. On the other hand, PRAEGENDE_JUGENDJAHRE can be re-encoded into a new binary column with value 0 refering to 'Mainstream' and 1 to ' Avantgarde' which are decided based on the original values of the feature from -1 to 15. We also re-encoded the CAMEO_INTL_2015 feature since it actually contains 2 variables (first and second digits); one being the level of wealth and the other being status.

Impute NaN values

We now can impute the NaN values with the median strategy.

```
In [66]: imputer = SimpleImputer(missing_values=np.nan, strategy='median')
         azdias_imputed = pd.DataFrame(imputer.fit_transform(azdias))
         customers_imputed = pd.DataFrame(imputer.fit_transform(customers))

In [67]: azdias_imputed.shape
Out[67]: (737288, 289)

In [68]: customers_imputed.shape
Out[68]: (159036, 288)

In [69]: list(set(customers_imputed)-set(azdias_imputed))
Out[69]: []
```

Scale features

```
In [70]: scaler = StandardScaler()
         azdias_scaled = pd.DataFrame(scaler.fit_transform(azdias_imputed))
         customers_scaled = pd.DataFrame(scaler.fit_transform(customers_imputed))

In [71]: azdias_scaled.shape
Out[71]: (737288, 289)

In [72]: customers_scaled.shape
Out[72]: (159036, 288)
```

After re-encoding features, we can then proceed to impute the NaN values and scale the features. Imputation is important so that the base count for all features are the same (not affected by NaN values) while keeping the changes to a minimum by using the median strategy, while feature scaling normalizes the ranges of all features.

Implementation - Part 1 (Unsupervised Learning)

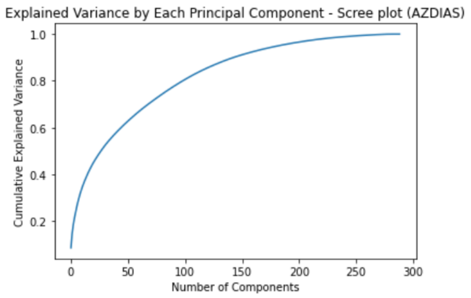
First and foremost, we will employ Principal Component Analysis (PCA) in order to reduce the dimensionality (the amount of features) of the datasets, while retaining the “principal components” which are linear combinations of existing features.

```
#Scree Azdias

pca = PCA()
azdias_pca = pca.fit_transform(azdias_scaled)

plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.title('Explained Variance by Each Principal Component - Scree plot (AZDIAS)')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')

plt.show()
```




```
def pca_150(df, n=150):
    pca = PCA(n_components=n)
    reduced_df = pca.fit_transform(df)
    reduced_df = pd.DataFrame(reduced_df)
    print('The variance in the data explained by the principal components after employing PCA is equal to ' + str(pca.explained_variance_ratio_.sum()))
    return reduced_df
```

```
reduced_azdias = pca_150(azdias_scaled, n=150)
reduced_customers = pca_150(customers_scaled, n=150)
```

```
The variance in the data explained by the principal components after employing PCA is equal to 0.9094487343051892
The variance in the data explained by the principal components after employing PCA is equal to 0.9094111973049335
```

```
reduced_azdias.shape
```

```
(737288, 150)
```

```
reduced_customers.shape
```

```
(159036, 150)
```

We used Scree plots in order to decide the ideal number of components for both AZDIAS and customers dataset. The tradeoff involved is that we need a number of component that is high enough to represent the variabilities of the data, but not too high so that our model is somewhat simpler. Additionally, as we can see from the plots, the curves became less steep after a certain point, or in other words, incremental explained variance does not increase by much with additional components. Therefore, judging from the plots, 150 components seems to be the ideal number for both datasets.

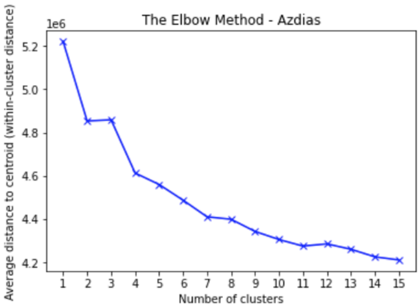
We then apply the PCA to the datasets, and calculate the explained variance ratio of the reduced datasets compared to the original ones. As seen from the printed statements, the explained variances are greater than 90%, which is a good tradeoff given we reduced the number of features to just slightly more than half of the original counts.

```python
def apply_kmeans(data, k):
    kmeans = KMeans(n_clusters=k)
    model = kmeans.fit(data)
    return abs(model.score(data))
```

```python
distances = []
possible_clusters = list(range(1,16))

for k in possible_clusters:
    distances.append(apply_kmeans(reduced_azdias.sample(20000), k))

plt.plot(possible_clusters, distances, linestyle='-', marker='x', color='blue')
plt.xticks(ticks=possible_clusters)
plt.xlabel('Number of clusters')
plt.ylabel('Average distance to centroid (within-cluster distance)')
plt.title('The Elbow Method - Azdias')
plt.show()
```



After PCA, we continued with the clustering of the dataset where we used k-Means. In order to get the ideal number of clusters, we used the elbow method, which is somewhat similar in essence to the scree plot; that is, choosing the right point of trade-off between explained variances and number of 'explainers', in elbow method's case: number of clusters.

From the plots, we can see that the elbow is located somewhere around 9-12 clusters. Thus, we pick 11 to be the ideal number of clusters.

```python
chosen_number_of_clusters = 11
kmeans = KMeans(n_clusters=chosen_number_of_clusters, random_state=101)
```

```python
# Azdias - general population clustering

azdias_preds = kmeans.fit_predict(reduced_azdias)
azdias_clustered = pd.DataFrame(azdias_preds, columns = ['Cluster'])
```

```python
# Customers - customer base clustering

customers_preds = kmeans.fit_predict(reduced_customers)
customers_clustered = pd.DataFrame(customers_preds, columns = ['Cluster'])
```

```
# Get cluster sizes for both datasets
azdias_clusters = pd.Series(azdias_preds)
num_pc = azdias_clusters.value_counts().sort_index()

customers_clusters = pd.Series(customers_preds)
num_cc = customers_clusters.value_counts().sort_index()
```

```
# Calculate % of each clusters
summary_df = pd.concat([num_pc, num_cc], axis=1).reset_index()
summary_df.columns = ['Cluster #','General Population','Customer Base']

summary_df['Perct of Total General Pop'] = ( summary_df['General Population'] / (summary_df['General Population'].sum()) * 100 ).round(2)
summary_df['Perct of Total Customer Base'] = ( summary_df['Customer Base'] / (summary_df['Customer Base'].sum()) * 100 ).round(2)
```

```
fig = plt.figure(figsize=(12,5))

ax = fig.add_subplot(111)

ax = summary_df['Perct of Total General Pop'].plot(x=summary_df['Cluster #'], width=-0.3, align='edge', color='orange', kind='bar', position=0)
ax = summary_df['Perct of Total Customer Base'].plot(kind='bar', color='green', width = 0.3, align='edge', position=1)

ax.set_xlabel('Clusters No.', fontsize=15)
ax.set_ylabel('Genpop (yellow) vs Customers (green)', fontsize=15)

ax.xaxis.set(ticklabels=range(11))
ax.tick_params(axis = 'x', which = 'major', labelsize = 13)
ax.margins(x=0.5,y=0.1)

plt.legend(('General Population (AZDIAS)', 'Customer Base (CUSTOMERS)'), fontsize=15)
plt.title(('% of General Population & Customer Base In Each Cluster'))

plt.show()
```

As seen above, clusters 2, 4, and 9 are the most underrepresented in the customer base, while on the other hand, clusters 1 and 5 are overrepresented.

Once we have settled on the number of clusters, we applied k-Means on both AZDIAS and customers and then compared the proportions of how each clusters make up each datasets. As seen from the bar chart, clusters 1 and 5 are more prevalent in the customer dataset than the AZDIAS, while on the other hand, clusters 2, 4, and 9 are underrepresented.

Implementation - Part 2 (Supervised Learning)

For the next step, we will employ classification algorithms in order to categorize individuals into two groups: those who are likely to convert into customers and those who are not. But firstly, we will explore the provided train dataset and apply the necessary pre-processing - this dataset contains the column "RESPONSE" which states whether or not a person became a customer of the company following the campaign.

```
mailout_train = pd.read_csv('train.csv', index_col=0)
```

```
mailout_train.head()
```

	LNR	AGER_TYP	AKT_DAT_KL	ALTER_HH	ALTER_KIND1	ALTER_KIND2	ALTER_KIND3	ALTER_KIND4	ALTERSKATEGORIE_FEIN	ANZ_HAUSHALTE_AKTIV	...	VK_DHT4A
0	1763	2	1.0	8.0	NaN	NaN	NaN	NaN	8.0	15.0	...	5.0
1	1771	1	4.0	13.0	NaN	NaN	NaN	NaN	13.0	1.0	...	1.0
2	1776	1	1.0	9.0	NaN	NaN	NaN	NaN	7.0	0.0	...	6.0
3	1460	2	1.0	6.0	NaN	NaN	NaN	NaN	6.0	4.0	...	8.0
4	1783	2	1.0	9.0	NaN	NaN	NaN	NaN	9.0	53.0	...	2.0

5 rows × 367 columns

```
mailout_train.shape
```

```
(42962, 367)
```

```
mailout_train_LNR = mailout_train['LNR']
```

```
pd.pivot_table(mailout_train, index='RESPONSE', aggfunc='count')
```

RESPONSE	AGER_TYP	AKT_DAT_KL	ALTERSKATEGORIE_FEIN	ALTERSKATEGORIE_GROB	ALTER_HH	ALTER_KIND1	ALTER_KIND2	ALTER_KIND3	ALTER_KIND4	ANREDE_KZ
0	42430	35544	34372	42430	35544	1967	746	170	40	42430
1	532	449	435	532	449	21	10	4	1	532

2 rows × 366 columns

As seen above, there is an imbalance between responses zero and one (non-customer and customer, respetively). We will resample the customers and balance the data.

```
Yes_Customer = mailout_train[mailout_train['RESPONSE']==1]
No_Customer = mailout_train[mailout_train['RESPONSE']==0]

Yes_Customer_balanced = resample(Yes_Customer, replace=True, n_samples=42430, random_state=1)

mailout_train_balanced = pd.concat([No_Customer, Yes_Customer_balanced])
```

```
pd.pivot_table(mailout_train_balanced, index='RESPONSE', aggfunc='count')
```

RESPONSE	AGER_TYP	AKT_DAT_KL	ALTERSKATEGORIE_FEIN	ALTERSKATEGORIE_GROB	ALTER_HH	ALTER_KIND1	ALTER_KIND2	ALTER_KIND3	ALTER_KIND4	ANREDE_KZ
0	42430	35544	34372	42430	35544	1967	746	170	40	42430
1	42430	35812	34605	42430	35812	1729	797	323	80	42430

As seen from the codes and outputs above, we noted that the dataset is imbalanced; there are much more of those whose RESPONSE column is equal to zero (did not convert into customer) than one, thus we conducted a resampling of the dataset in order to balance the two groups - we randomly sampled the 532 individuals whose RESPONSE is equal to one 42430 times to get a new dataset that contains 84860 evenly split rows.

```
# Separate and drop labels
train_y_label = mailout_train_balanced['RESPONSE']
mailout_train_balanced.drop(labels=['RESPONSE'], axis=1, inplace=True)
```

```
# Combine all cleaning processes done on azdias and customers earlier to one function, for ease

def cleaning_function(df, attributes, to_drop_nonexistent, columns_to_drop_missing, nan_threshold=0.2):

    # drop features nonexistent
    df.drop(labels=to_drop_nonexistent, axis=1, inplace=True)

    print('after drop features not described' + str(df.shape))

    for column in df.columns:
        if df[column].dtype == np.int64:
            df[column] = df[column].astype(np.float64)


    unknown_series = pd.Series(attributes['Missing Values'].values, index=attributes['Attributes'])

    # convert missing values to NaN
    for column in df.columns:
        isin = ast.literal_eval(unknown_series[column])
        df[column] = df[column].mask(df[column].isin(isin), other=np.nan)

    print('after missing values' + str(unknown_series.shape))
    print('after missing values' + str(df.shape))

    # drop columns with higher than threshold % NaN
    drop_columns = []
    for column in columns_to_drop_missing:
        if column not in to_drop_nonexistent:
            drop_columns.append(column)

    df.drop(labels=drop_columns, axis=1, inplace=True)

    print('after drop lower threshold' + str(df.shape))

    #re-encoding

    cat_feat_df = []
    for cat_feat in categorical_features_attributes:
        if cat_feat in df.columns:
            cat_feat_df.append(cat_feat)

    mixed_feat_df = []
    for mixed_feat in mixed_features_attributes:
        if mixed_feat in df.columns:
            mixed_feat_df.append(mixed_feat)
```

```
    binary_num_attributes = []
    binary_non_num_attributes = []
    multi_level_attributes = []

    for attribute in cat_feat_df:
        dtype = df[attribute].dtype
        count = len(df[attribute].value_counts())

        if count > 2:
            multi_level_attributes.append(attribute)
        else:
            if dtype == 'object':
                binary_non_num_attributes.append(attribute)
            else:
                binary_num_attributes.append(attribute)

    df['OST_WEST_KZ'] = df['OST_WEST_KZ'].map({'W': 1, 'O': 2})

    list_columns_to_add = []

    for column in multi_level_attributes:

        if len(df[column].value_counts()) < 10:
            list_columns_to_add.append(pd.get_dummies(df[column], prefix=column))


    df.drop(multi_level_attributes, axis=1, inplace=True)

    print('after drop multilevel' + str(df.shape))

    list_columns_to_add.append(df)

    df = pd.concat(list_columns_to_add, axis=1)

    print('after adding cols to add' + str(df.shape))

    ### df mixed features re-encoding ###

    df['PRAEGENDE_JUGENDJAHRE'] = df['PRAEGENDE_JUGENDJAHRE'].apply(lambda x: encode_pj(x))
    df.drop('PRAEGENDE_JUGENDJAHRE', axis=1, inplace=True)

    df.drop(['LP_LEBENSPHASE_FEIN', 'LP_LEBENSPHASE_GROB'], axis=1, inplace=True)

    df['WEALTH_LEVEL'] = df['CAMEO_INTL_2015'].apply(lambda x: np.floor(pd.to_numeric(x)/10))
    df['STATUS'] = df['CAMEO_INTL_2015'].apply(lambda x: pd.to_numeric(x)%10)
    df.drop('CAMEO_INTL_2015', axis=1, inplace=True)

    print('final' + str(df.shape))

    return df
```

```
# plug in values
not_described_features = to_drop_nonexistent

missing_perct_column = mailout_train_balanced.isnull().mean(axis=0)

columns_to_drop_missing = missing_perct_column[missing_perct_column > 0.2].index
```

```
# apply function
train = cleaning_function(mailout_train_balanced, attributes, not_described_features, columns_to_drop_missing, nan_threshold=0.2)
```

Next, we separate the RESPONSE column (dependent/explained variable) from the rest of the features (independent/explainer variables) for the classifier algorithms purposes. After that, we applied the same cleaning processes which we applied to AZDIAS and customers datasets, but we combined all of them into one function. We also applied this function to the test dataset later on.

```
# Impute and Scale
imputer = SimpleImputer(missing_values=np.nan, strategy='median')
train_imputed = pd.DataFrame(imputer.fit_transform(train))

scaler = StandardScaler()
train_imputed_scaled = pd.DataFrame(scaler.fit_transform(train_imputed))
```

After cleaning the data, we also applied imputation and scaling on the train data.

```
# training grid search function
def gridsearch_classifier(classifier, param_grid, X_train, y_train):
    # using StratifiedKFold

    start = time.time()

    grid = GridSearchCV(estimator=classifier, param_grid=param_grid, scoring='roc_auc', cv=5)
    grid.fit(X_train, y_train)

    end = time.time()
    print('/n')
    print('Time Taken:' + str(end-start))
    print(grid.best_score_)
    return grid.best_estimator_
```

1. Benchmark Model - Logistic Regression

Our benchmark model is the Logistic Regression classifier since this is the standard model with 1 as “converted into new customer” score and 0 as the “not converted into new customer” score.

```
logistic = LogisticRegression(random_state=101)
print(gridsearch_classifier(logistic, {}, train_imputed_scaled, train_y_label))
```

```
/n
Time Taken:17.068783044815063
0.8079572243568297
LogisticRegression(random_state=101)
```

> Logistic Regression: Time Taken: 17.1 seconds, AUC = 0.808

Once the training data has been pre-processed, we can continue with the classifiers. First, we applied the logistic regression, which we have decided as the benchmark model for this project. It performed quite well, with AUC of 0.808 and quick processing time of 17 seconds.

As for the models we will be testing, we picked five potential models:

- Decision Tree Classifier
- Random Forest Classifier
- AdaBoost Classifier
- Gradient Boosting Classifier

- Multilayer Perceptron (Neural Network) Classifier

```
tree = DecisionTreeClassifier(random_state=101)
rf = RandomForestClassifier(random_state=101)
adaboost = AdaBoostClassifier(random_state=101)
gbc = GradientBoostingClassifier(random_state=101)
mlp_nn = MLPClassifier(random_state=101, alpha=1, max_iter=100)
```

```
print(gridsearch_classifier(tree, {}, train_imputed_scaled, train_y_label))
```

```
/n
Time Taken:56.231369972229004
0.9846861787133946
DecisionTreeClassifier(random_state=101)
```

```
print(gridsearch_classifier(rf, {}, train_imputed_scaled, train_y_label))
```

```
/n
Time Taken:184.84636998176575
0.9933737501908706
RandomForestClassifier(random_state=101)
```

```
print(gridsearch_classifier(adaboost, {}, train_imputed_scaled, train_y_label))
```

```
/n
Time Taken:133.06420803070068
0.7859658744471563
AdaBoostClassifier(random_state=101)
```

```
print(gridsearch_classifier(gbc, {}, train_imputed_scaled, train_y_label))
```

```
/n
Time Taken:607.3762629032135
0.8886038998171919
GradientBoostingClassifier(random_state=101)
```

```
print(gridsearch_classifier(mlp_nn, {}, train_imputed_scaled, train_y_label))
```

```
/n
Time Taken:637.5171508789062
0.9903483737671325
MLPClassifier(alpha=1, max_iter=100, random_state=101)
```

After getting the same scores and time results for these models using the same grid search function we prepared, we noticed that the Decision Tree, Random Forest, and Multilayer Perceptron algorithms are the best performing models, but the results (both in terms of duration and AUC) are somewhat too good to be true, and might be caused by overfitting. Other than those three, Gradient Boosting Classifier performed the best despite taking more time to fit. We decided to use this model as our final model.

```
#Define the possible hyperparameters (loss type and max depth) for the GB
param_grid = {'loss': ['deviance','log_loss', 'exponential'],
              'max_depth': list(range(3,5))
             }

gbc_best_estimator = gridsearch_classifier(gbc, param_grid, train_imputed_scaled, train_y_label)
```

```
/n
Time Taken:2269.597023010254
0.9492743007031754
```

```
gbc_best_estimator
```

```
GradientBoostingClassifier(max_depth=4, random_state=101)
```

Using the same grid search function, we tried to tune the max_depth parameter of the model for the Gradient Boosting Classifier. We ended up with an AUC score of 0.949 which is much better compared to the model's first score, as well as the benchmark model's (logistic regression) score.

```
mailout_test = pd.read_csv('test.csv', index_col=0)
```

```
mailout_test.head()
```

	LNR	AGER_TYP	AKT_DAT_KL	ALTER_HH	ALTER_KIND1	ALTER_KIND2	ALTER_KIND3	ALTER_KIND4	ALTERSKATEGOI
0	1754	2	1.0	7.0	NaN	NaN	NaN	NaN	
1	1770	-1	1.0	0.0	NaN	NaN	NaN	NaN	
2	1465	2	9.0	16.0	NaN	NaN	NaN	NaN	
3	1470	-1	7.0	0.0	NaN	NaN	NaN	NaN	
4	1478	1	1.0	21.0	NaN	NaN	NaN	NaN	

5 rows × 366 columns

```
mailout_test.shape
```

```
(42833, 366)
```

```
test_LNR = mailout_test['LNR']
```

```
# apply function for cleaning - use same parameters as train data in terms of columns to drop
test = cleaning_function(mailout_test,
                         attributes,
                         not_described_features,
                         columns_to_drop_missing,
                         nan_threshold=0.2)
```

```
# Impute and Scale
imputer = SimpleImputer(missing_values=np.nan, strategy='median')
test_imputed = pd.DataFrame(imputer.fit_transform(test))

scaler = StandardScaler()
test_imputed_scaled = pd.DataFrame(scaler.fit_transform(test_imputed))
```

We then moved to the test dataset. We applied the same cleaning function, imputation, and feature scaling processes to it, and then proceeded with the Gradient Boosting Classifier we trained.

Classification with Chosen Model ()

```
# Calculate prediction probabilities for the TEST set with previously trained GradientBoostingClassifier estimator
test_gbc_preds = gbc_best_estimator.predict_proba(test_imputed_scaled)
```

CSV File for Kaggle Submission

Columns: the LNR id column + prediction of response column

```
submission = pd.DataFrame({'LNR':test_LNR.astype(np.int32), 'RESPONSE':test_gbc_preds[:, 1]})
submission.to_csv('submission.csv', index=False)
submission.head()
```

	LNR	RESPONSE
0	1754	0.636801
1	1770	0.635613
2	1465	0.179187
3	1470	0.042028
4	1478	0.059016

```
len(submission[submission['RESPONSE']>0.5])
```

```
6133
```

Once done, we populated the RESPONSE column using the predict_proba method in order to predict each individual's probability of getting put into class = 1, which in this specific case means converting as a customer. This is the final submission csv file required by the Kaggle competition. Additionally, the count of the RESPONSE column with value greater than 0.5 is 6133, which means we predicted that there are 6133 individuals in the test dataset who will become customers, or approximately 14% of the individuals in the test dataset.

Conclusion

I enjoyed working on this project since it is an actual real world situation/datasets that I am working on, which gave me a clear picture of real world problems which data scientists work on in real life. Personally, I think the most difficult part of the project is the data cleaning and pre-processing step, but I believe it is the one I appreciated the most since it really tells me a lot about how to treat data and not just rely on the final methods/models to give the answers.

Some considerations on how future work can be done to solve this project better include implementing neural networks using PyTorch or TensorFlow, instead of just sklearn’s MLP model, since neural networks are the state-of-the-art model for most machine learning tasks currently.

On the unsupervised learning part, the differences between the two datasets can be studied more, such as what features are more significantly represented in each imbalanced clusters, and why these might come to be. The results of this analysis can help many aspects of the business such as identifying untapped demographics which exist in the population but not the customer base, as well as fine tuning communication materials so that they fit the current demographics of the customer base.