Capstone Project

Machine Learning Engineer
Nanodegree

Mark Anthony B. Dungo

February, 2020

Customer Segmentation Report for Arvato Financial Services

Definition

Project Overview

A showcase how 3D clusters' visualization works with customer segmentation using k-means clustering and a trained supervised learning model that can predict whether a group of people will respond on an offer based on their characteristics from a demographics data.

Problem Statement

I. Customer Segmentation Report
   A. The first is figuring out how to have a better analysis which among the general population might respond to an offer.
   B. I manually compare the descriptions of `azdias` and `customers` to get their high mean gap columns.
   C. I also figure out how k-means clustering might help us grouping data items.

II. Supervised Learning Model
   A. Get the most relevant features correlated to `mailout_train['RESPONSE']`.
   B. Find the best classifier that will work well on our demographics data.
   C. Finally, to have a model that can predict with good accuracy and ROC AUC scores.

Metrics

I. Customer Segmentation Report
   A. In general, we must set the number of clusters that will give a k-means model the highest average silhouette score.
   B. But that's not always the case, we can adjust the `n_clusters` not just to have a high average silhouette score but is also reasonable for the dataset to have a better visualization.

II. Supervised Learning Model
   A. I choose to improve the classifier that has the highest Area Under the Receiver Operating Characteristic Curve (ROC AUC) score on my multiple classifiers and minimum correlation targets' test.
   B. In training, I clone and only save the model if it has a closer accuracy and ROC AUC scores on the best scores.

Analysis

Data Exploration

We have these demographics datasets loaded in on the "Arvato Project Workbook.ipynb" notebook:

- `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).

Here's the imports and declared variables. In case you encounter some variables that's not declared, consider checking "Arvato Project Workbook.ipynb", all of the codes are written there.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import os
import datetime

import sklearn
from sklearn.model_selection import train_test_split
from sklearn.kernel_approximation import Nystroem
```

```
from sklearn.calibration import CalibratedClassifierCV
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.base import clone
from sklearn import ensemble

# magic word for producing visualizations in notebook
%matplotlib inline

data_dir = 'arvato_data'
extra_attrs = ['CUSTOMER_GROUP', 'ONLINE_PURCHASE', 'PRODUCT_GROUP']
now = datetime.datetime.now()

if not os.path.exists(data_dir):
    os.makedirs(data_dir)
```

I load “Udacity_AZDIAS_052018.csv” and assign it into azdias and “Udacity_CUSTOMERS_052018.csv” into customers with the following code:

```
azdias = pd.read_csv('../../data/Term2/capstone/arvato_data/Udacity_AZDIAS_052018.csv', sep=';', index_col=False)
customers = pd.read_csv('../../data/Term2/capstone/arvato_data/Udacity_CUSTOMERS_052018.csv', sep=';', index_col=False)
```

I used this code to quickly get the means of both columns of azdias and customers excluding its three extra columns ('CUSTOMER_GROUP', 'ONLINE_PURCHASE', and 'PRODUCT_GROUP'):

```
azdias_mean = azdias.describe().loc['mean', :]

customers_no_extra = customers.loc[:, ~customers.columns.isin(extra_attrs)]
customers_no_extra_mean = customers_no_extra.describe().loc['mean', :]
```

Display the high mean gap attributes between azdias_mean and customers_no_extra_mean:

```
customers_azdias_high_gap_attrs = []

for i, item in customers_no_extra_mean.iteritems():
    azdias_mean_item = azdias_mean[i]
    gap = abs(azdias_mean_item - item)

    if (gap >= min_attr_gap and i != 'LNR'):
        print("gap: {} | attr: {} | azdias' val: {} | customers' val: {}".format(gap, i, azdias_mean_item, item))
        customers_azdias_high_gap_attrs.append(i)

print('\ncustomers_azdias_high_gap_attrs:', customers_azdias_high_gap_attrs)
```

Output:

```
gap: 3.369137995421216 | attr: ALTERSKATEGORIE_FEIN | azdias' val: 13.70071656633889 | customers' val: 10.331578570917674
gap: 3.3213997272377185 | attr: ANZ_HAUSHALTE_AKTIV | azdias' val: 8.28726319522149 | customers' val: 4.965863467983771
gap: 4.544008062817511 | attr: EINGEZOGENAM_HH_JAHR | azdias' val: 2003.7290607321315 | customers' val: 1999.185052669314
```

gap: 5.0802070439637035 | attr: EXTSEL992 | azdias' val: 33.338392359997975 | customers' val: 38.41859940396168
gap: 97.7857999361321 | attr: GEBURTSJAHR | azdias' val: 1101.178532597414 | customers' val: 1003.3927326612819
gap: 47.529776748614154 | attr: KBA13_ANZAHL_PKW | azdias' val: 619.7014391008134 | customers' val: 667.2312158494276
gap: 3.5589335696132007 | attr: LP_LEBENSPHASE_FEIN | azdias' val: 14.622637124351426 | customers' val: 18.181570693964627
gap: 3.906072643824456 | attr: PRAEGENDE_JUGENDJAHRE | azdias' val: 8.154345555142887 | customers' val: 4.248272911318431

customers_azdias_high_gap_attrs: ['ALTERSKATEGORIE_FEIN', 'ANZ_HAUSHALTE_AKTIV', 'EINGEZOGENAM_HH_JAHR', 'EXTSEL992', 'GEBURTSJAHR', 'KBA13_ANZAHL_PKW', 'LP_LEBENSPHASE_FEIN', 'PRAEGENDE_JUGENDJAHRE']
Length: 8

Display the descriptions of both high mean gap columns between azdias and customers:

```
azdias[customers_azdias_high_gap_attrs].describe()
```

	ALTERSKATEGORIE_FEIN	ANZ_HAUSHALTE_AKTIV	EINGEZOGENAM_HH_JAHR	EXTSEL992	GEBURTSJAHR	KBA13_ANZAHL_PKW	LP_LEBENSPHAS
count	628274.000000	798073.000000	817722.000000	237068.000000	891221.000000	785421.000000	886367
mean	13.700717	8.287263	2003.729061	33.338392	1101.178533	619.701439	14
std	5.079849	15.628087	7.058204	14.537408	976.583551	340.034318	12
min	0.000000	0.000000	1900.000000	1.000000	0.000000	0.000000	0
25%	11.000000	1.000000	1997.000000	23.000000	0.000000	384.000000	4
50%	14.000000	4.000000	2003.000000	34.000000	1943.000000	549.000000	11
75%	17.000000	9.000000	2010.000000	43.000000	1970.000000	778.000000	27
max	25.000000	595.000000	2018.000000	56.000000	2017.000000	2300.000000	40

```
customers[customers_azdias_high_gap_attrs].describe()
```

	ALTERSKATEGORIE_FEIN	ANZ_HAUSHALTE_AKTIV	EINGEZOGENAM_HH_JAHR	EXTSEL992	GEBURTSJAHR	KBA13_ANZAHL_PKW	LP_LEBENSPHAS
count	139810.000000	141725.000000	145056.000000	106369.000000	191652.000000	140371.000000	188439
mean	10.331579	4.965863	1999.185053	38.418599	1003.392733	667.231216	18
std	4.134828	14.309694	6.178099	13.689466	974.531081	340.481722	15
min	0.000000	0.000000	1986.000000	1.000000	0.000000	5.000000	0
25%	9.000000	1.000000	1994.000000	29.000000	0.000000	430.000000	0
50%	10.000000	1.000000	1997.000000	36.000000	1926.000000	593.000000	16
75%	13.000000	4.000000	2004.000000	53.000000	1949.000000	828.000000	36
max	25.000000	523.000000	2018.000000	56.000000	2017.000000	2300.000000	40

Getting the high mean gap attributes between purchasers_mean (mean for customers that purchased) and azdias_mean:

```
# select all the customers that purchased
purchasers = customers.loc[customers['ONLINE_PURCHASE'] > 0]

# exclude the extra attributes
purchasers_no_extra = purchasers.loc[:, ~purchasers.columns.isin(extra_attrs)]

# get the mean
purchasers_no_extra_mean = purchasers_no_extra.describe().loc['mean', :]

purchasers_azdias_high_gap_attrs = []
```

```
for i, item in purchasers_no_extra_mean.iteritems():
    azdias_mean_item = azdias_mean[i]
    gap = abs(azdias_mean_item - item)

    if (gap >= min_attr_gap and i != 'LNR'):
        print("gap: {} | attr: {} | azdias' val: {} | purchasers' val: {}".format(gap, i, azdias_mean_item, item))
        purchasers_azdias_high_gap_attrs.append(i)

print('\npurchasers_azdias_high_gap_attrs:', purchasers_azdias_high_gap_attrs)
```

Output:

```
gap: 3.6350069812520687 | attr: ALTER_HH | azdias' val: 10.864126194476851 | purchasers' val: 14.49913317572892
gap: 3.579048127505976 | attr: ANZ_HAUSHALTE_AKTIV | azdias' val: 8.28726319522149 | purchasers' val: 4.708215067715514
gap: 3.286891678930277 | attr: ANZ_STATISTISCHE_HAUSHALTE | azdias' val: 7.599356199244931 | purchasers' val: 4.312464520314654
gap: 84.9084817411615 | attr: GEBURTSJAHR | azdias' val: 1101.178532597414 | purchasers' val: 1186.0870143385755
gap: 51.940587134645625 | attr: KBA13_ANZAHL_PKW | azdias' val: 619.7014391008134 | purchasers' val: 671.6420262354591
gap: 4.035533348923826 | attr: LP_LEBENSPHASE_FEIN | azdias' val: 14.622637124351426 | purchasers' val: 18.658170473275252

purchasers_azdias_high_gap_attrs: ['ALTER_HH', 'ANZ_HAUSHALTE_AKTIV', 'ANZ_STATISTISCHE_HAUSHALTE', 'GEBURTSJAHR',
'KBA13_ANZAHL_PKW', 'LP_LEBENSPHASE_FEIN']
```

Display the common high mean gap attributes' descriptions between customers_azdias_high_gap_attrs and purchasers_azdias_high_gap_attrs:

```
# get the attributes that are in both arrays
common_high_gap_attrs = np.intersect1d(customers_azdias_high_gap_attrs, purchasers_azdias_high_gap_attrs)

for i, row in attributes_df.iterrows():
    for item in np.nditer(common_high_gap_attrs):
        if (row['Attribute'] == item):
            print('   Attribute: "{}" | Description: "{}"'.format(item, row.Description))
            print(" >> for customers:\n{}".format(customers[item].describe()))
            print(" >> azdias:\n{}\n".format(azdias[item].describe()))
```

Output:

```
   Attribute: "GEBURTSJAHR" | Description: "year of birth"
 >> for customers:
count    191652.000000
mean       1003.392733
std         974.531081
min           0.000000
25%           0.000000
50%        1926.000000
75%        1949.000000
max        2017.000000
Name: GEBURTSJAHR, dtype: float64
 >> azdias:
count    891221.000000
mean       1101.178533
std         976.583551
min           0.000000
25%           0.000000
50%        1943.000000
```

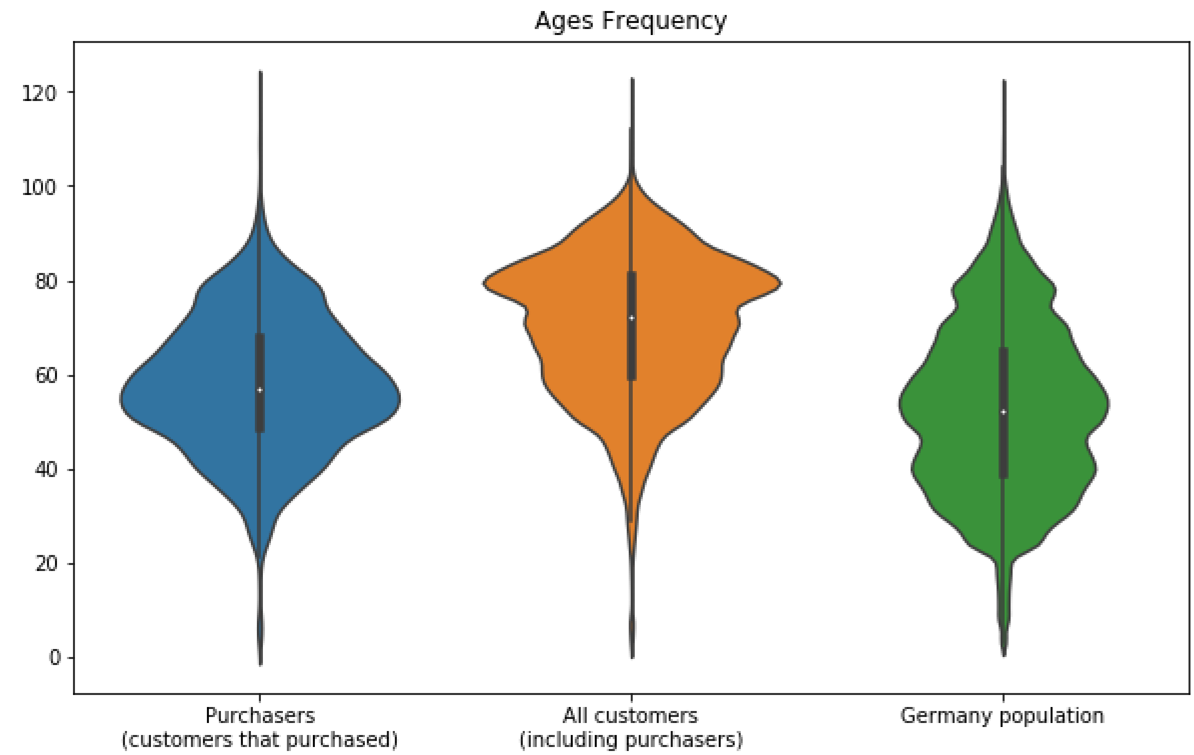
```
75%        1970.000000
max        2017.000000
Name: GEBURTSJAHR, dtype: float64

   Attribute: "LP_LEBENSPHASE_FEIN" | Description: "lifestage fine"
 >> for customers:
count    188439.000000
mean         18.181571
std          15.009985
min           0.000000
25%           0.000000
50%          16.000000
75%          36.000000
max          40.000000
Name: LP_LEBENSPHASE_FEIN, dtype: float64
 >> azdias:
count    886367.000000
mean         14.622637
std          12.616883
min           0.000000
25%           4.000000
50%          11.000000
75%          27.000000
max          40.000000
Name: LP_LEBENSPHASE_FEIN, dtype: float64

   Attribute: "ANZ_HAUSHALTE_AKTIV" | Description: "number of households known in this building"
 >> for customers:
count    141725.000000
mean          4.965863
std          14.309694
min           0.000000
25%           1.000000
50%           1.000000
75%           4.000000
max         523.000000
Name: ANZ_HAUSHALTE_AKTIV, dtype: float64
 >> azdias:
count    798073.000000
mean          8.287263
std          15.628087
min           0.000000
25%           1.000000
50%           4.000000
75%           9.000000
max         595.000000
Name: ANZ_HAUSHALTE_AKTIV, dtype: float64

   Attribute: "KBA13_ANZAHL_PKW" | Description: "number of cars in the PLZ8"
 >> for customers:
count    140371.000000
mean        667.231216
std         340.481722
min           5.000000
25%         430.000000
50%         593.000000
75%         828.000000
max        2300.000000
Name: KBA13_ANZAHL_PKW, dtype: float64
 >> azdias:
count    785421.000000
mean        619.701439
std         340.034318
min           0.000000
```

```
25%       384.000000
50%       549.000000
75%       778.000000
max      2300.000000
Name: KBA13_ANZAHL_PKW, dtype: float64
```

Exploratory Visualization

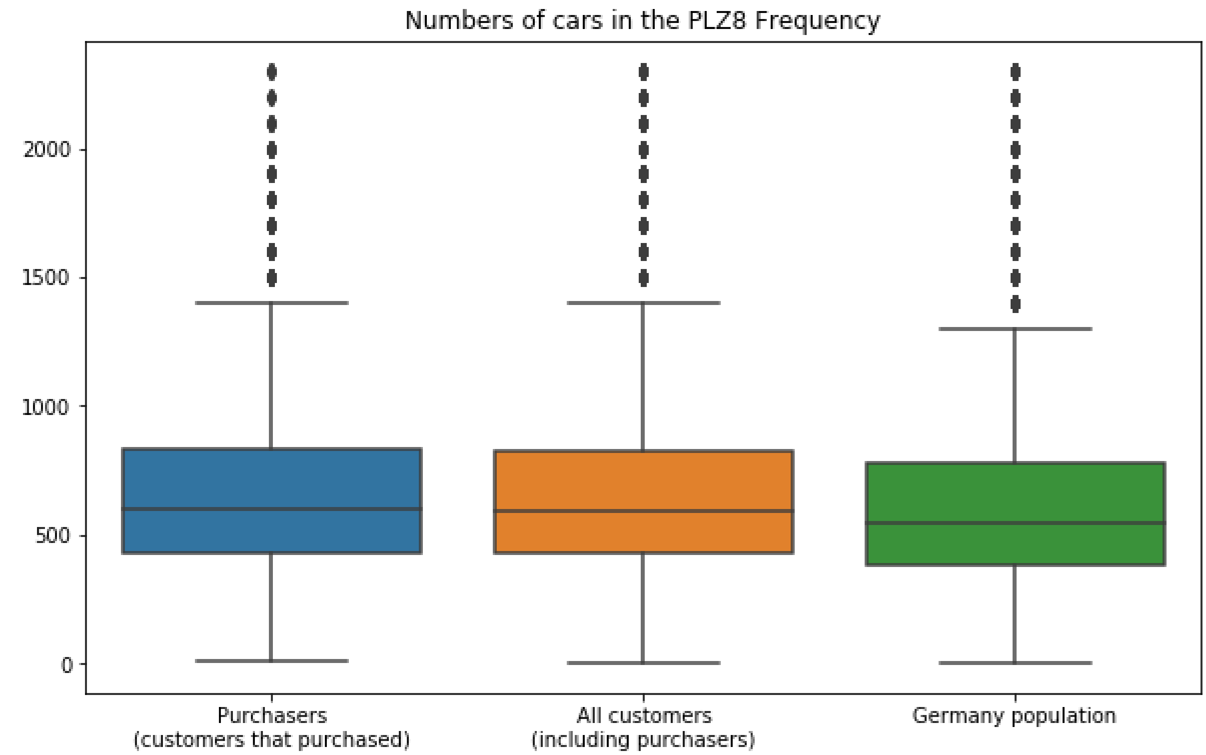
I get all the ages from purchasers, customers and azdias to display their “Ages Frequency”:

```
xlabels = [
    'Purchasers\n(customers that purchased)',
    'All customers\n(including purchasers)',
    'Germany population'
]

def get_ages_from_data(data):
    birth_years = data['GEBURTSJAHR']
    birth_years = birth_years[~(birth_years == 0)] # remove the years that have a value of 0
    return birth_years.apply(lambda x: now.year - x) # convert birth years to ages

purchasers_age = get_ages_from_data(purchasers)
customers_age = get_ages_from_data(customers)
azdias_age = get_ages_from_data(azdias)

plt.figure(figsize=(10,6))
plt.title("Ages Frequency")

ax = sns.violinplot(data=[purchasers_age, customers_age, azdias_age])\
    .set_xticklabels(xlabels)
```



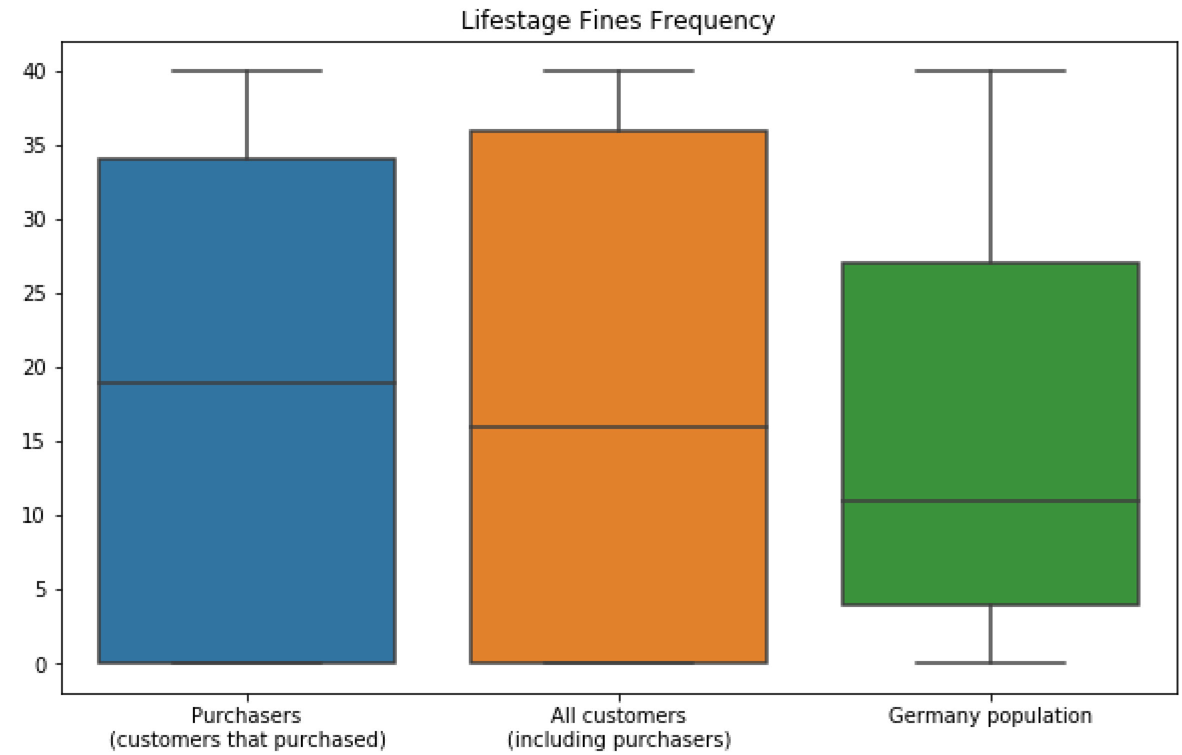
Displaying their “Numbers of cars in the PLZ8 Frequency”:

```
purchasers_car_num = purchasers['KBA13_ANZAHL_PKW']
customers_car_num = customers['KBA13_ANZAHL_PKW']
azdias_car_num = azdias['KBA13_ANZAHL_PKW']

plt.figure(figsize=(10,6))
plt.title("Numbers of cars in the PLZ8 Frequency")

ax = sns.boxplot(data=[purchasers_car_num, customers_car_num, azdias_car_num])\
    .set_xticklabels(xlabels)
```



And their Lifestage Fines Frequency:

```
customers_lifestage_fine = customers['LP_LEBENSPHASE_FEIN']
azdias_lifestage_fine = azdias['LP_LEBENSPHASE_FEIN']

plt.figure(figsize=(10,6))
plt.title("Lifestage Fines Frequency")

ax = sns.boxplot(data=[purchasers_lifestage_fine, customers_lifestage_fine, azdias_lifestage_fine])\
    .set_xticklabels(xlabels)
```



K-Means Clustering

Since the "year of birth", "number of cars in the PLZ8" and "lifestage fine" are the top three attributes of the highest mean gap columns between our demographics datas, they are good candidates for our k-means model.

There are other very common and useful segmentation methods as well. One of them is called RFM which stands for Recency, Frequency, Monetary Value.

```
# num of rows to clear for the training data
X_row_num = 2000

def select_kmeans_x(data):
    attrs = ['GEBURTSJAHR', 'KBA13_ANZAHL_PKW', 'LP_LEBENSPHASE_FEIN']

    if 'ONLINE_PURCHASE' in data:
        attrs.append('ONLINE_PURCHASE')

    # select few attributes
    X = data[attrs][:X_row_num] # select few rows to proceed fast

    # fill all NaN with 0 to fix the
    # "ValueError: Input contains NaN, infinity or a value too large for dtype('float64')."
    X = X.fillna(0)

    X = X[X['GEBURTSJAHR'] > 0] # only select the rows that have a birth year greater than 0
    birth_years = X['GEBURTSJAHR']
    X['GEBURTSJAHR'] = birth_years.apply(lambda x: now.year - x) # convert birth years to ages
    return X
```

```
X = select_kmeans_x(azdias)
print('X dimension:', X.shape)

# display the datasets
print(X.head())
```

Output:

```
X dimension: (1029, 3)
  GEBURTSJAHR  KBA13_ANZAHL_PKW  LP_LEBENSPHASE_FEIN
1          24             963.0                 21.0
2          41             712.0                  3.0
3          63             596.0                  0.0
4          57             435.0                 32.0
5          77            1300.0                  8.0
```

Getting average silhouette scores with Std Scaling based on numbers of clusters.

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

x = list(range(2, 12))
y_std = []

for n_clusters in x:
    print("n_clusters =", n_clusters)

    kmeans = KMeans(n_clusters=n_clusters, init='k-means++', n_init=10, max_iter=300)
    kmeans.fit(X)

    clusters = kmeans.predict(X)
    silhouette_avg = silhouette_score(X, clusters)

    y_std.append(silhouette_avg)
    print("The average silhouette_score is: {} with Std Scaling".format(silhouette_avg))
```

Output:

```
n_clusters = 2
The average silhouette_score is: 0.5895090131239678 with Std Scaling
n_clusters = 3
The average silhouette_score is: 0.5114030166331367 with Std Scaling
n_clusters = 4
The average silhouette_score is: 0.5153760510006951 with Std Scaling
n_clusters = 5
The average silhouette_score is: 0.5065242907083939 with Std Scaling
n_clusters = 6
The average silhouette_score is: 0.4920834405346492 with Std Scaling
n_clusters = 7
The average silhouette_score is: 0.5069787110889967 with Std Scaling
n_clusters = 8
The average silhouette_score is: 0.505152664680305 with Std Scaling
n_clusters = 9
The average silhouette_score is: 0.4946813163896397 with Std Scaling
n_clusters = 10
The average silhouette_score is: 0.47367943054505923 with Std Scaling
n_clusters = 11
The average silhouette_score is: 0.46315254933338923 with Std Scaling
```

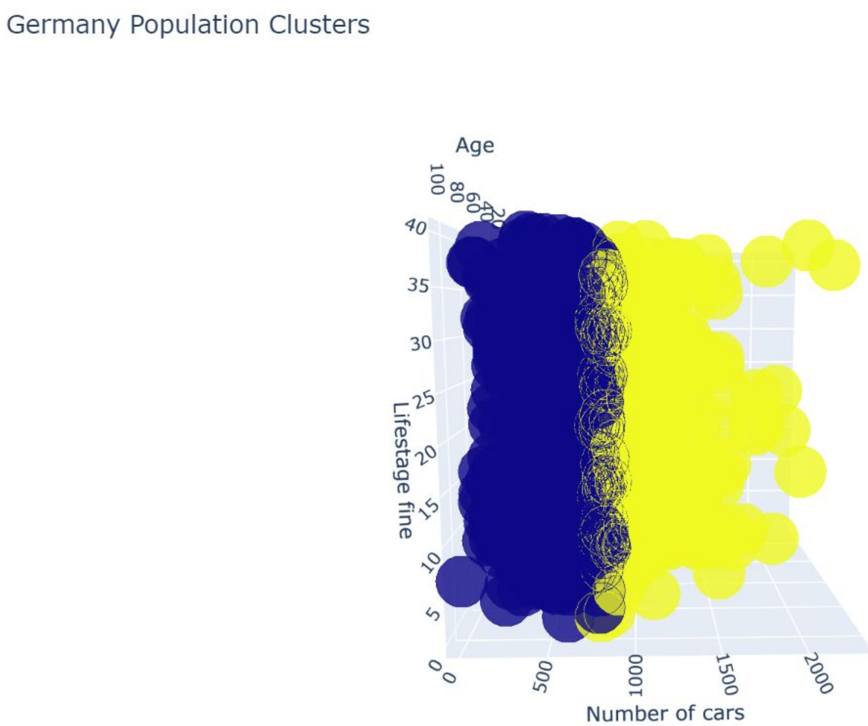
In practice, it's good to select an n_clusters that will give your k-means model the highest average silhouette_score with your dataset.

```
# # uncomment to install plotly
# conda install -c plotly plotly=4.5.0

import plotly as py
import plotly.graph_objs as go

def clusters_3d(n_clusters, data, attrs, title):
    kmeans = KMeans(n_clusters=n_clusters, init='k-means++', n_init=10, max_iter=300)
    kmeans.fit(data)
    clusters = kmeans.predict(data) # [0, n < n_clusters, n < n_clusters, etc...]

    data['clusters'] = clusters

    # display the data
    print(data.head())

    trace1 = go.Scatter3d(
        x = data[attrs[0]['index']],
        y = data[attrs[1]['index']],
        z = data[attrs[2]['index']],
        mode = 'markers',
        marker = {
            'color': data['clusters'],
            'size': 20,
            'line': {
                'color': data['clusters'],
                'width': 12
            },
            'opacity': 0.8
        }
    )
    data = [trace1]
    layout = go.Layout(
        height = 600,
        title = title,
        scene = {
            'xaxis': { 'title': attrs[0]['title'] },
            'yaxis': { 'title': attrs[1]['title'] },
            'zaxis': { 'title': attrs[2]['title'] }
        }
    )
    fig = go.Figure(data=data, layout=layout)

    py.offline.iplot(fig)
    return kmeans


attrs = [{
    'index': 'GEBURTSJAHR',
    'title': 'Age'
}, {
    'index': 'KBA13_ANZAHL_PKW',
    'title': 'Number of cars'
```

```
}, {
    'index': 'LP_LEBENSPHASE_FEIN',
    'title': 'Lifestage fine'
}]

kmeans_model = clusters_3d(2, X, attrs, 'Germany Population Clusters')
```

Output:

	GEBURTSJAHR	KBA13_ANZAHL_PKW	LP_LEBENSPHASE_FEIN	clusters
1	24	963.0	21.0	1
2	41	712.0	3.0	0
3	63	596.0	0.0	0
4	57	435.0	32.0	0
5	77	1300.0	8.0	1



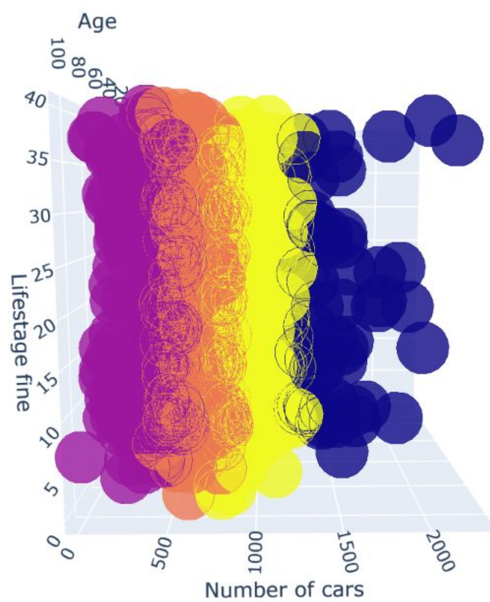
However, we can still increase the n_clusters to give our result not just a high average silhouette_score but also a reasonable number of groups for a better view.

```
kmeans_model = clusters_3d(4, X, attrs, 'Germany Population Clusters')
```

Output:

	GEBURTSJAHR	KBA13_ANZAHL_PKW	LP_LEBENSPHASE_FEIN	clusters
1	24	963.0	21.0	3
2	41	712.0	3.0	2
3	63	596.0	0.0	2
4	57	435.0	32.0	2
5	77	1300.0	8.0	0

Germany Population Clusters



Custom Visualization

A 3D graph of customers where we can see which bought online or not.

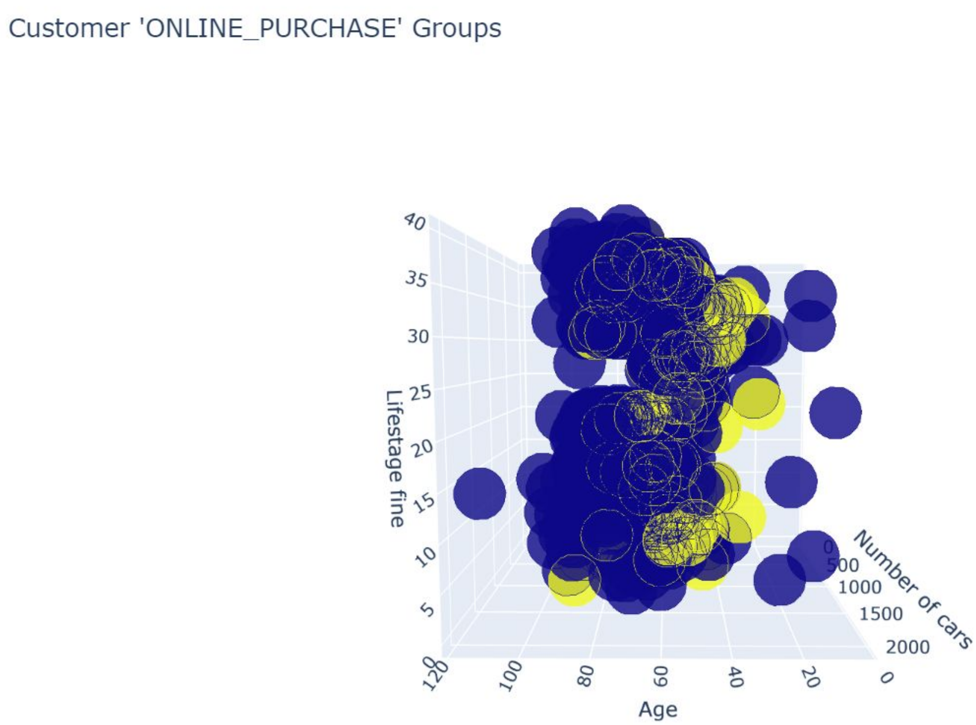
```
customers_X = select_kmeans_x(customers)

# display the dataset
print(customers_X.head(), '\n')

trace1 = go.Scatter3d(
    x = customers_X[attrs[0]['index']],
    y = customers_X[attrs[1]['index']],
    z = customers_X[attrs[2]['index']],
    mode = 'markers',
    marker = {
        'color': customers_X['ONLINE_PURCHASE'],
        'size': 20,
        'line': {
            'color': customers_X['ONLINE_PURCHASE'],
            'width': 12
        },
        'opacity': 0.8
    }
)
data = [trace1]
layout = go.Layout(
    height = 600,
    title = "Customer 'ONLINE_PURCHASE' Clusters",
    scene = {
        'xaxis': { 'title': attrs[0]['title'] },
        'yaxis': { 'title': attrs[1]['title'] },
        'zaxis': { 'title': attrs[2]['title'] }
```

```
    }
)
fig = go.Figure(data=data, layout=layout)

# display a description of the dataset
print(customers_X.describe())

py.offline.iplot(fig)
```

Output:

	GEBURTSJAHR	KBA13_ANZAHL_PKW	LP_LEBENSPHASE_FEIN	ONLINE_PURCHASE
4	60	513.0	31.0	0
6	78	1300.0	20.0	0
7	82	481.0	20.0	0
8	106	428.0	6.0	0
9	61	1106.0	28.0	0

	GEBURTSJAHR	KBA13_ANZAHL_PKW	LP_LEBENSPHASE_FEIN	ONLINE_PURCHASE
count	733.000000	733.000000	733.000000	733.000000
mean	68.724420	692.826739	23.821282	0.107776
std	15.881233	376.281666	12.900713	0.310309
min	3.000000	0.000000	0.000000	0.000000
25%	58.000000	456.000000	13.000000	0.000000
50%	71.000000	598.000000	25.000000	0.000000
75%	80.000000	889.000000	37.000000	0.000000
max	118.000000	2100.000000	40.000000	1.000000



Only 10.78% (0.107776 * 100, yellow) of our cleared 733 customers which bought online. Most of them are 50 to 60 years olds, as what’s in the previous “Ages Frequency”.

That's how 3D visualization works in grouping. You can select other x-y-z attributes that correlate to ‘ONLINE_PURCHASE’ before displaying figures. This is just one of many tricks on how you can visualize a dataset.

Algorithms and Techniques

Directly train a supervised model with the common classifier called `sklearn.svm.SVC` and fit it according to all the present features on our encoded dataset takes an hour and will just have an ROC AUC score around 0.50.

I try to use the following techniques to have a better model for our dataset that contains a lot of unstable `NaN` and zeros:

- Encode all the invalid feature values.
- Selecting the best features that correlate to ‘RESPONSE’.
- Optimize the dataset to speed up the training process.
- Choose the best classifier for our demographics dataset.
- Only save the cloned model if it has good training scores.

I create a script that evaluates multiple classifiers and iterates that process with a different number of relevant features to help me find the right classifier and the exact relevant features.

I tried to use the Nystroem method to speed up the training process but once I try to get another prediction’s ROC AUC scores using the same model, it’s having unstable results for the same test dataset like an overfitting curve, sometimes the scores are around 0.40 which is very low, sometimes they’re around 0.70, sometimes around 0.50 and later figured out that the output after fitting and transforming the data with the Nystroem method is always different from the other same transformed data.

Once I found the best classifier for our demographics dataset and select the most relevant features correlate to ‘RESPONSE’, the final outcome might have a better ROC AUC score.

I will then repeatedly clone and train the best performing model with “Udacity_MAILOUT_052018_TRAIN.csv” again and again but only save it when it has a closer accuracy and ROC AUC scores to the model’s current high scores.

Benchmark

I declared multiple classifiers and test them one by one, iterating the process with different relative features based on the listed minimum correlation targets.

At the end, the classifiers that have the highest ROC AUC scores have used the class `AdaBoostClassifier` and `GradientBoostingClassifier` both have default parameters. I choose the `AdaBoostClassifier` and repeatedly clone, train and save the high performer to “arvato_data/model.joblib.pkl”.

Methodology

Data Pre-processing

Since our demographics dataset contains a lot of NaN, zeros and invalid values, I encode them all until the full dataset becomes valid to be fitted into a classification model and use a filter method called Pearson correlation to get the relevant features correlate to ‘RESPONSE’.

Implementation

Encode non-numeric characters

I load in the “Udacity_MAILOUT_052018_TRAIN.csv” with the following:

```
mailout_train = pd.read_csv('../../data/Term2/capstone/arvato_data/Udacity_MAILOUT_052018_TRAIN.csv', sep=';', index_col=False)
```

Whenever there's an incompatible value I test it with this:

```
for item in row.iteritems():
        if (item[1] == 'W'):
            print(i, item)
            break
        if (item[1] == '1992-02-10 00:00:00'):
            print(i, item)
            break
        if (item[1] == 'D19_UNBEKANNT'):
            print(i, item)
            break
        if (item[1] == '4A'):
            print(i, item)
            break
        if (item[1] == 'XX'):
            print(i, item)
            break
        if (item[1] == 'X'):
            print(i, item)
            break
```

These array and functions do the work to clear and encode some incompatible values on our dataset:

```
# string attributes to encode to their unique numeric values
str_attrs = [
    'OST_WEST_KZ', # flag strings
```

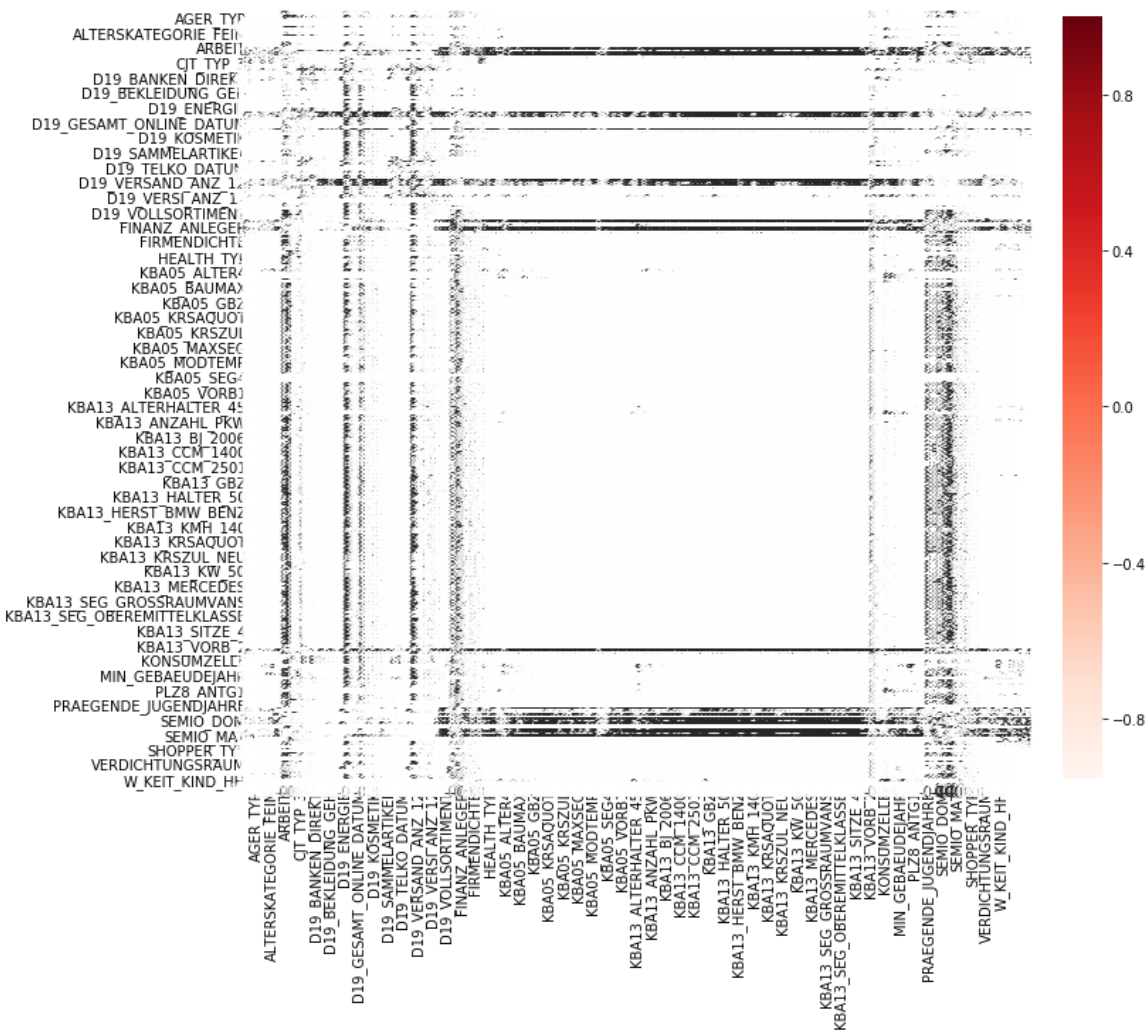
```
    'D19_LETZTER_KAUF_BRANCHE', # string values
    'CAMEO_DEU_2015' # 2-char strings
]

# function cleans train and test features
def clear_features(data):
    # fill all NaN with 0 to fix the
    # "ValueError: Input contains NaN, infinity or a value too large for dtype('float64')."
    data = data.fillna(0)
    print('All NaN values are now converted to 0.')

    for attr in str_attrs:
        if attr in data.columns:
            print(" - Previous column:\n{}".format(data[attr].head()))
            data = encode_data_str_col(data, attr)
            print(" - New column:\n{}\n".format(data[attr].head()))

    if 'EINGEFUEGT_AM' in data.columns:
        print(" - Previous column:\n{}".format(data['EINGEFUEGT_AM'].head()))
        # encode date strings with numbers of days in-between
        data['EINGEFUEGT_AM'] = data['EINGEFUEGT_AM'].apply(convert_date_to_days)
        print(" - New column:\n{}\n".format(data['EINGEFUEGT_AM'].head()))

    if (data.columns.isin(['CAMEO_INTL_2015', 'CAMEO_DEUG_2015']).any()):
        # replace all 'XX' and 'X' values with NaN
        if 'CAMEO_INTL_2015' in data.columns:
            data['CAMEO_INTL_2015'].replace(['XX'], [0], inplace=True)
            print("'XX' values are now replaced by NaN.")
        if 'CAMEO_DEUG_2015' in data.columns:
            data['CAMEO_DEUG_2015'].replace(['X'], [0], inplace=True)
            print("'X' values are now replaced by NaN.")
        print()

    return data

# encode column string values with numerical values
def encode_data_str_col(data, attr):
    unique_vals = data[attr].unique()
    unique_str = [s for s in unique_vals if isinstance(s, str)]

    nums = range(1, len(unique_str) + 1)
    data[attr].replace(unique_str, nums, inplace=True)

    print('  >> Number of unique column strings:', len(unique_str))
    print('  >> {} have been encoded to {}.'.format(unique_str, nums))
    return data

def convert_date_to_days(x):
    return (now - datetime.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')).days if isinstance(x, str) else x
```

Getting Relevant Features

I divide `mailout_train` columns to have our training features and their corresponding class labels and use Pearson correlation to get the highly correlated features to ‘RESPONSE’.

```
X = clear_features(mailout_train)
y = mailout_train['RESPONSE']
del X['LNR']
# don't forget to `del X['RESPONSE']` on an actual training

# # not effective for computing the pairwise correlation of columns
# feature_map_nystroem = Nystroem(n_components=X.shape[1])
# X_2 = feature_map_nystroem.fit_transform(X)
# X = pd.DataFrame(X_2, columns=X.columns) # convert back to DataFrame

# using Pearson correlation
plt.figure(figsize=(12,10))
cor = X.corr()
sns.heatmap(cor, annot=True, cmap=plt.cm.Reds)
plt.plot()
plt.show()
```

Output: # it prints the encoding details of the invalid feature values and displays this:



Test how many relevant features will produce for the listed minimum correlation targets.

```
# minimum correlation targets to test
min_cor_targets = [
    # custom
    0.010, 0.011, 0.012, 0.013, 0.0136,
    0.015, 0.0155, 0.01584, 0.016, 0.0165, 0.017, 0.020,
    0.030, 0.031, 0.035,
    # and the others...

#     # correlation target description values
#     0.005359, 0.004708, 0.000055, 0.002178, 0.004217, 0.007067, 0.040392,
]

# correlation with output variable
cor_target = abs(cor["RESPONSE"]).drop('RESPONSE')

print('Correlation target (w/o "RESPONSE") description:\n{}\n'.format(cor_target.describe()))

for min_cor_target in min_cor_targets:
    print(' - Minimum correlation target: {:.10f}'.format(min_cor_target))

    # selecting highly correlated features
    relevant_features = cor_target[cor_target >= min_cor_target]

    print(' - Number of relevant features:', len(relevant_features))
    print(' - Relevant features:\n{}\n'.format(relevant_features))
```

Output:

```
Correlation target (w/o "RESPONSE") description:
count    363.000000
mean       0.005359
std        0.004708
min        0.000055
25%        0.002178
50%        0.004217
75%        0.007067
max        0.040392
Name: RESPONSE, dtype: float64

 - Minimum correlation target: 0.0100000000
 - Number of relevant features: 46
 - Relevant features:
AGER_TYP                 0.013432
AKT_DAT_KL               0.012428
ANZ_TITEL                0.010388
# OTHERS…
Name: RESPONSE, dtype: float64

 - Minimum correlation target: 0.0110000000
 - Number of relevant features: 36
 - Relevant features:
AGER_TYP                 0.013432
AKT_DAT_KL               0.012428
CJT_GESAMTTYP            0.011814
# OTHERS…
```

Name: RESPONSE, dtype: float64

- Minimum correlation target: 0.0120000000
- Number of relevant features: 27
- Relevant features:

AGER_TYP 0.013432
AKT_DAT_KL 0.012428
CJT_TYP_3 0.012673
others…
Name: RESPONSE, dtype: float64

(And many more hidden output items…)

My script that evaluates multiple classifiers with different parameters and compute accuracy & ROC AUC scores and iterates the process with a different length of features based on the listed minimum correlation targets.

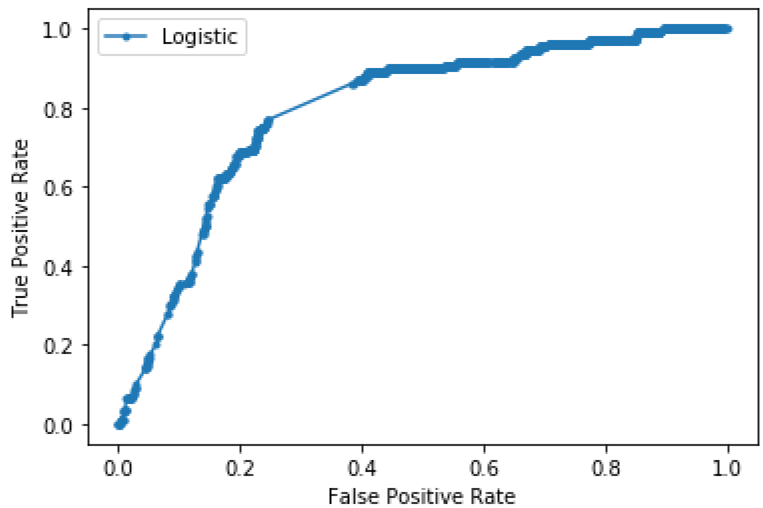
```
# represents the proportion of the dataset to include in the test split
test_size = 0.2

# # number of data items to train
# n_items = 20000

# list of classifiers
clfs = [
#     sklearn.svm.SVC(gamma='scale', probability=True),
#     sklearn.svm.SVC(kernel='poly', gamma='scale', probability=True),

    # wrap with CalibratedClassifierCV to support .predict_proba()
    CalibratedClassifierCV(sklearn.svm.LinearSVC(dual=False), cv=5),

    # .predict_proba() is only available for log loss and modified Huber loss
    sklearn.linear_model.SGDClassifier(loss='log'),
    sklearn.linear_model.SGDClassifier(loss='modified_huber'),

    ensemble.AdaBoostClassifier(),
    ensemble.BaggingClassifier(),
    ensemble.ExtraTreesClassifier(n_estimators=100),
    ensemble.GradientBoostingClassifier(),
    ensemble.RandomForestClassifier(n_estimators=100),

    sklearn.linear_model.LogisticRegression(solver='lbfgs', max_iter=7600),
    sklearn.neighbors.KNeighborsClassifier()
]

# minimum correlation targets
min_cor_targets = [
    0.015, 0.0155, 0.01584, 0.016, 0.0165, 0.017, 0.020,
    0.030, 0.031, 0.035,
    # and the others...
]

models_arr = []
print('test_size: {}\n'.format(test_size))
```

```
for i, min_cor_target in enumerate(min_cor_targets):
    models_arr.append([])

    # selecting highly correlated features
    relevant_features = cor_target[cor_target >= min_cor_target]

    # new instance of our features and class labels
    X = clear_features(mailout_train[relevant_features.index]) # clear_features(mailout_train[relevant_features.index][:n_items])
    y = mailout_train['RESPONSE'] # mailout_train['RESPONSE'][:n_items]

    print('Number of relevant features:', X.shape[1])

    # split training and test features and class labels
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)

    print('X_train dimension: {} | y_train dimension: {}'.format(X_train.shape, y_train.shape))
    print('X_test dimension: {} | y_test dimension: {}\n'.format(X_test.shape, y_test.shape))

    for j, clf in enumerate(clfs):
        model = clone(clf)

        # start training
        model.fit(X_train, y_train)

        preds = model.predict(X_test)
        probs = model.predict_proba(X_test)
        probs = probs[:, 1]

        models_arr[i].append(model)
        print('models_arr[{}][{}]: {}'.format(i, j, model))

        accuracy = accuracy_score(y_test, preds) * 100
        roc_auc = roc_auc_score(y_test, probs)

        print('Predictions:\n{}'.format(preds[:10]))
        print('Probabilities:\n{}'.format(probs[:10]))
        print('Correct labels:\n{}'.format(y_test[:10]))
        print('Minimum correlation target: {:.10f}'.format(min_cor_target))
        print('Accuracy: {}% | ROC AUC: {}'.format(accuracy, roc_auc))

        # calculate roc curve
        fpr, tpr, _ = roc_curve(y_test, probs)
        # plot the roc curve for the model
        plt.plot(fpr, tpr, marker='.', label='Logistic')
        # axis labels
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        # show the legend
        plt.legend()
        # show the plot
        plt.show()
```

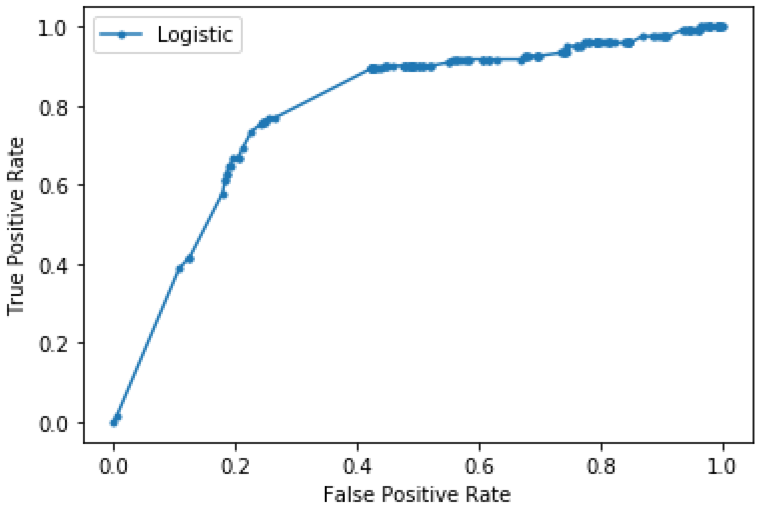
```
print('-' * 125, '\n')
```

The output of the above code is a list of all different classifier evaluations.
Here are the top 3 high performing classifiers on my test:

```
models_arr[1][3]: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None, learning_rate=1.0,
            n_estimators=50, random_state=None)
Predictions:
[0 0 0...]
Probabilities:
[0.47347266 0.46870559 0.47747548...]
Correct labels:
16415    0
38600    0
29606    0...
Name: RESPONSE, dtype: int64
Minimum correlation target: 0.0155000000
Accuracy: 98.74316303968347% | ROC AUC: 0.7961298806172112
```




```
models_arr[6][3]: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,  learning_rate=1.0,
            n_estimators=50, random_state=None)
Predictions:
[0 0 0...]
Probabilities:
[0.47224556 0.46842721 0.46816637...]
Correct labels:
13964    0
41441    0
33790    0...
Name: RESPONSE, dtype: int64
Minimum correlation target: 0.0200000000
Accuracy: 98.59187710927499% | ROC AUC: 0.7888601440623073
```



models_arr[1][6]: GradientBoostingClassifier(criterion='friedman_mse', init=None,
learning_rate=0.1, loss='deviance', max_depth=3,
max_features=None, max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=100,
n_iter_no_change=None, presort='auto',
random_state=None, subsample=1.0, tol=0.0001,
validation_fraction=0.1, verbose=0,
warm_start=False)

Predictions:

[0 0 0...]

Probabilities:

[0.00395712 0.00227169 0.02253164...]

Correct labels:

16415 0

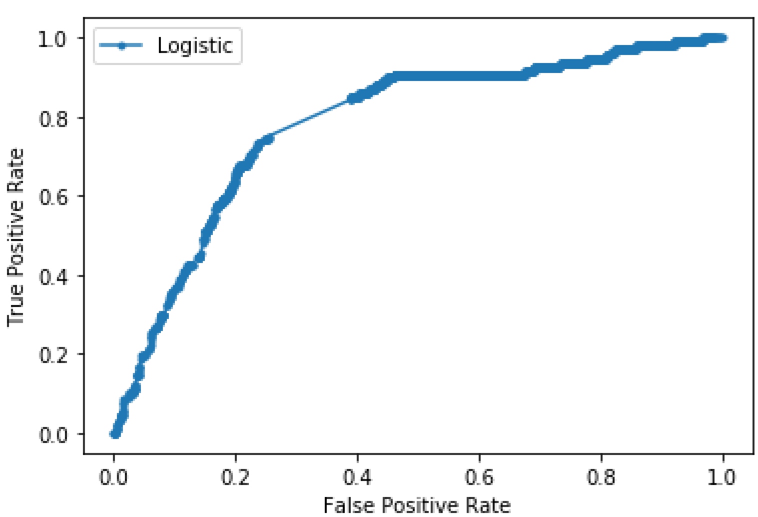
38600 0

29606 0...

Name: RESPONSE, dtype: int64

Minimum correlation target: 0.0155000000

Accuracy: 98.74316303968347% | ROC AUC: 0.7836427028088784



Refinement

- Training with.sklearn.svm.SVC without explicitly declaring the parameter gamma takes longer to complete. Sometimes, it hangs on a large dataset.
- Training with all just encoded features will just result to have around 0.5 of ROC AUC score because our training data features contain a lot of inconsistent NaN and zeros, which means it's performing poorly and its probability predictions are almost random.
- I tried to use the Nystroem method to speed up the training process but the computed pairwise correlation of columns are not that effective to be used for getting the relevant features because the output after fitting and transforming the data using that method is always different from the other same transformed data. You will notice that I've commended out this kind of code on the "Arvato Project Workbook.ipynb" notebook.

```
# feature_map_nystroem = Nystroem(n_components=X.shape[1])
# X_2 = feature_map_nystroem.fit_transform(X)
# X = pd.DataFrame(X_2, columns=X.columns) # convert back to DataFrame
# print(X.head(), '\n')
```

- The outcome by using Pearson correlation to get our target relevant features to increase our ROC AUC score is quite effective.
- One thing that I notice in scoring, when the ROC AUC score is high, the accuracy score is a little bit lower from its stable score.

Model Training

I assigned the best performing classifier to a variable named model.

```
from sklearn.externals import joblib
filename = data_dir + '/model.joblib.pkl'

# # uncomment to load the model
# model = joblib.load(filename)

# store to model
model = models_arr[1][3]

# store the minimum correlation target
min_cor_target = 0.0155000000

# selecting highly correlated features
relevant_features = cor_target[cor_target >= min_cor_target]

# new instance of our features and class labels
X = clear_features(mailout_train[relevant_features.index])
y = mailout_train['RESPONSE']
```

```
print('Minimum correlation target: {:.10f}'.format(min_cor_target))
print('Number of relevant features:', X.shape[1])
print('Relevant features:', relevant_features.index)
```

Output:

```
All NaN values are now converted to 0.
Minimum correlation target: 0.0155000000
Number of relevant features: 9
Relevant features: Index(['D19_BANKEN_DIREKT', 'D19_KONSUMTYP', 'D19_KONSUMTYP_MAX',
    'D19_SOZIALES', 'FINANZ_VORSORGER', 'KBA05_CCM4', 'KBA05_KW3',
    'RT_KEIN_ANREIZ', 'RT_SCHNAEPPCHEN'],
   dtype='object')
```

And repeatedly clone and train it with the full encoded dataset, assign to model and save the trained model as .joblib.pkl file in the data_dir whenever it has an acceptable accuracy and ROC AUC scores.

```
test_size = 0.3
# accuracy_current = 0.9899914655908139
# roc_auc_current = 0.8407732158546246

# deduction for the current score to save the model
deduction = 0.012

# number of passes through the entire training dataset
epochs = range(5000) # or infinity() and just interrupt the kernel to cancel training

print('epochs:', epochs)
print('deduction:', deduction)
print('test_size:', test_size)
print('model: {}'.format(model))
print('# accuracy_current = {}\n# roc_auc_current = {}\n'.format(accuracy_current, roc_auc_current))

for i in epochs:
#     print('Epoch:', i + 1)

    # split training and test features and class labels
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)

    # start training with full dataset
    model2 = clone(model)

    model2.fit(X, y)

    preds = model2.predict(X_test)
    probs = model2.predict_proba(X_test)
    probs = probs[:, 1]

    accuracy_new = accuracy_score(y_test, preds)
    roc_auc_new = roc_auc_score(y_test, probs)

    if (roc_auc_new > roc_auc_current - deduction and accuracy_new > accuracy_current - deduction):
        model = model2
        _ = joblib.dump(model, filename, compress=9)

        print('Predictions:\n{}'.format(preds[:3]))
        print('Probabilities:\n{}'.format(probs[:3]))
        print('Correct labels:\n{}'.format(y_test[:3]))

        print('New accuracy: {} | New ROC AUC: {}'.format(accuracy_new, roc_auc_new))
```

```
    if (roc_auc_new > roc_auc_current):
        roc_auc_current = roc_auc_new
        print('roc_auc_current updated!')
    if (accuracy_new > accuracy_current):
        accuracy_current = accuracy_new
        print('accuracy_current updated!')

    print('# accuracy_current = {}\n# roc_auc_current = {}\n'.format(accuracy_current, roc_auc_current))
```

I sometimes tweak the variables such as `test_size`, `deduction` & `epochs` and reset `accuracy_current` and `roc_auc_current` by decreasing them. I will use the latest model to predict the probabilities of my submission's RESPONSE for "Udacity+Arvato: Identify Customer Segments" competition through Kaggle.

Results

Model Evaluation and Validation

Average Scores

Accuracy: 0.9876169638284996 | ROC AUC: 0.786644256174256

The one of the first parts of my training outputs is as follows. The very first is the training details including the cloned model, followed by an item representing the result of the saved trained model that has accuracy and ROC AUC scores of no lower than the previous model's scores deducted by the `deduction` value.

```
epochs: <generator object infinity at 0x000001F8B72F98C8>
deduction: 0.012
test_size: 0.3
model: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None, learning_rate=1.0,
          n_estimators=50, random_state=None)
# accuracy_current = 0.9899914655908139
# roc_auc_current = 0.8393468449881631

Predictions:
[0 0 0]
Probabilities:
[0.47139289 0.47749122 0.47749122]
Correct labels:
40928    0
16486    0
40740    0
Name: RESPONSE, dtype: int64
New accuracy: 0.987586313911087 | New ROC AUC: 0.8324780029853092
# accuracy_current = 0.9899914655908139
# roc_auc_current = 0.8393468449881631

Predictions:
[0 0 0]
Probabilities:
[0.46768132 0.46750847 0.4826985 ]
```

Correct labels:
15770   0
4397   0
15012   0
Name: RESPONSE, dtype: int64
New accuracy: 0.988362169291644 | New ROC AUC: 0.8274982337703116
accuracy_current = 0.9899914655908139
roc_auc_current = 0.8393468449881631

Predictions:
[0 0 0]
Probabilities:
[0.48417027 0.48525351 0.47394594]
Correct labels:
7610   0
24301   1
10059   0
Name: RESPONSE, dtype: int64
New accuracy: 0.9895259523624796 | New ROC AUC: 0.8324728915837587
accuracy_current = 0.9899914655908139
roc_auc_current = 0.8393468449881631

Predictions:
[0 0 0]
Probabilities:
[0.47749122 0.48215826 0.46324332]
Correct labels:
28823   0
3462   0
42333   0
Name: RESPONSE, dtype: int64
New accuracy: 0.9872759717588642 | New ROC AUC: 0.8328781925343811
accuracy_current = 0.9899914655908139
roc_auc_current = 0.8393468449881631

Predictions:
[0 0 0]
Probabilities:
[0.47027919 0.47749122 0.46929192]
Correct labels:
34612   0
18460   0
39611   0
Name: RESPONSE, dtype: int64
New accuracy: 0.9888276825199783 | New ROC AUC: 0.8273846497537161
accuracy_current = 0.9899914655908139
roc_auc_current = 0.8393468449881631

Predictions:
[0 0 0]
Probabilities:
[0.47619097 0.47781631 0.46668009]
Correct labels:
25392   0
35486   0
26239   0
Name: RESPONSE, dtype: int64
New accuracy: 0.9884397548296997 | New ROC AUC: 0.8298086142045873
accuracy_current = 0.9899914655908139
roc_auc_current = 0.8393468449881631

```
Predictions:
[0 0 0]
Probabilities:
[0.47142176 0.47232859 0.48150258]
Correct labels:
37215    0
17356    0
7941     0
Name: RESPONSE, dtype: int64
New accuracy: 0.9871208006827528 | New ROC AUC: 0.8303854417907423
# accuracy_current = 0.9899914655908139
# roc_auc_current = 0.8393468449881631

# and others…

Predictions:
[0 0 0]
Probabilities:
[0.48448735 0.47253786 0.46722583]
Correct labels:
8117     0
28883    0
13611    1
Name: RESPONSE, dtype: int64
New accuracy: 0.9867328729924743 | New ROC AUC: 0.8327355711709424
# accuracy_current = 0.9899914655908139
# roc_auc_current = 0.8393468449881631
```

On this result item, you can see that the 24301 correct label has the probability of 0.48525351 which is good but the prediction is wrong:

```
Predictions:
[0 0 0]
Probabilities:
[0.48417027 0.48525351 0.47394594]
Correct labels:
7610     0
24301    1
10059    0
Name: RESPONSE, dtype: int64
New accuracy: 0.9895259523624796 | New ROC AUC: 0.8324728915837587
# accuracy_current = 0.9899914655908139
# roc_auc_current = 0.8393468449881631
```

Getting the Final Scores

```
print('model: {}'.format(model))

preds = model.predict(X)
probs = model.predict_proba(X)
probs = probs[:, 1]

accuracy = accuracy_score(y, preds)
roc_auc = roc_auc_score(y, probs)

print('Predictions:\n{}'.format(preds[:5]))
```

```
    print('Probabilities:\n{}'.format(probs[:5]))
    print('Correct labels:\n{}'.format(y_test[:5]))
    print('Accuracy: {} | ROC AUC: {}\n'.format(accuracy, roc_auc))
```

Output:

```
model: AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None, learning_rate=1.0,
          n_estimators=50, random_state=None)
Predictions:
[0 0 0 0 0]
Probabilities:
[0.48568698 0.46721738 0.47466352 0.48525351 0.48355633]
Correct labels:
2744     0
10001    0
1084     0
6637     0
12529    0
Name: RESPONSE, dtype: int64
Accuracy: 0.9876169638284996 | ROC AUC: 0.786644256174256
```

Kaggle Competition "Udacity+Arvato: Identify Customer Segments"

Apply machine learning techniques to predict customers using data provided by Arvato Financial Solutions.

My Kaggle Profile Competitions: https://www.kaggle.com/the5ervant/competitions

Here's my submission:

Name	Wait time	Execution time	Score
Arvato_Capstone_Submission.csv	0 seconds	0 seconds	0.78691

Justification

The best performing classifier is AdaBoostClassifier followed by GradientBoostingClassifier. The correlation between mailout_train['RESPONSE'] and the other mailout_train features is very low, so we just select a few of the features. On my test, 4 to 10 features will also work well.

Splitting the data to have a test dataset instead of evaluating and predicting with the full data features, just makes me very late to notice that the model isn't improving because splitting data just gives me unstable result scores based on the random selection of sklearn.model_selection.train_test_split.

Conclusion

Reflection

The initial challenge for me what kind of Machine Learning techniques are suitable for our 'RESPONSE' prediction and to get what relevant features to improve ROC AUC score since our dataset columns are very complex. I've figured out how to use Pearson correlation to get the highly correlated features.

In the end, I feel that I'm still not happy with the final ROC AUC scores of my model. I think it should improve well if I choose a different approach which will also take time, especially by retraining a model.

Improvement

I. For Customer Segmentation Report, we can use Pearson correlation to get the top 3 correlated relevant features to our `customers['ONLINE_PURCHASE']` which we can use for the 3D clusters' visualization of the general population.

II. For the Supervised Learning Model

  A. I trained with `deduction = 0.04` and still got the same average scores of my model that's trained with `deduction = 0.012` proving that I just waste a lot of training hours by just storing a model if it reaches scores that's not lower than the current minus my `deduction`.

  B. The reason why I'm not saving the clone if it has low training scores is, I think it's becoming unlearned when I do that, just double check if this technique is effective or not.

  C. I notice that my repetitive training is just resulting in unstable scores depending on the random output train and test subsets of `sklearn.model_selection.train_test_split`. In our case with our demographics data, not splitting data for test subsets and directly evaluate and predict with our full data features *(just make sure to explicitly declare the parameter `gamma` of `sklearn.svm.SVC` if you're going to use it)* will give us a stable or consistent result if our model is improving or not.

  D. I'm hoping that generally using a classifier all with default parameters will do the work to make itself adjusts for the dataset, but we never know what will be the result by explicitly adjusting and tweaking them one by one, it might be better for our situation or not. But first, just do a quick search what parameters to adjust for a specific classifier.

There are still other techniques that's not mentioned here to increase accuracy and ROC AUC scores of a supervised model even if you have a complex dataset. We keep learning every time, there's a lot of things we can implement to improve our project.