Arvato Project Workbook

August 29, 2020

1 Capstone Project: Create a Customer Segmentation Report for Arvato Financial Services

In this project, you will analyze demographics data for customers of a mail-order sales company in Germany, comparing it against demographics information for the general population. You'll use unsupervised learning techniques to perform customer segmentation, identifying the parts of the population that best describe the core customer base of the company. Then, you'll apply what you've learned on a third dataset with demographics information for targets of a marketing campaign for the company, and use a model to predict which individuals are most likely to convert into becoming customers for the company. The data that you will use has been provided by our partners at Bertelsmann Arvato Analytics, and represents a real-life data science task.

If you completed the first term of this program, you will be familiar with the first part of this project, from the unsupervised learning project. The versions of those two datasets used in this project will include many more features and has not been pre-cleaned. You are also free to choose whatever approach you'd like to analyzing the data rather than follow pre-determined steps. In your work on this project, make sure that you carefully document your steps and decisions, since your main deliverable for this project will be a blog post reporting your findings.

In [ ]:
```
# import libraries here; add more as necessary
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
# from yellowbrick.cluster.elbow import kelbow_visualizer
from sklearn.cluster import KMeans
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline


pd.set_option('display.max_columns', 500)
```

```
        # magic word for producing visualizations in notebook
        %matplotlib inline
```

```
In [ ]: # %whos DataFrame
```

1.1 Part 0: Get to Know the Data

There are four data files associated with this project:

- `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).

Each row of the demographics files represents a single person, but also includes information outside of individuals, including information about their household, building, and neighborhood. Use the information from the first two files to figure out how customers ("CUSTOMERS") are similar to or differ from the general population at large ("AZDIAS"), then use your analysis to make predictions on the other two files ("MAILOUT"), predicting which recipients are most likely to become a customer for the mail-order company.

The "CUSTOMERS" file contains three extra columns ('CUSTOMER_GROUP', 'ONLINE_PURCHASE', and 'PRODUCT_GROUP'), which provide broad information about the customers depicted in the file. The original "MAILOUT" file included one additional column, "RESPONSE", which indicated whether or not each recipient became a customer of the company. For the "TRAIN" subset, this column has been retained, but in the "TEST" subset it has been removed; it is against that withheld column that your final predictions will be assessed in the Kaggle competition.

Otherwise, all of the remaining columns are the same between the three data files. For more information about the columns depicted in the files, you can refer to two Excel spreadsheets provided in the workspace. One of them is a top-level list of attributes and descriptions, organized by informational category. The other is a detailed mapping of data values for each feature in alphabetical order.

In the below cell, we've provided some initial code to load in the first two datasets. Note for all of the `.csv` data files in this project that they're semicolon (`;`) delimited, so an additional argument in the `read_csv()` call has been included to read in the data properly. Also, considering the size of the datasets, it may take some time for them to load completely.

You'll notice when the data is loaded in that a warning message will immediately pop up. Before you really start digging into the modeling and analysis, you're going to need to perform some cleaning. Take some time to browse the structure of the data and look over the informational spreadsheets to understand the data values. Make some decisions on which features to keep, which features to drop, and if any revisions need to be made on data formats. It'll be a good idea to create a function with pre-processing steps, since you'll need to clean all of the datasets before you work with them.

Examine azdias dataframe

```
In [ ]: # load in the data
        azdias = pd.read_csv('../../data/Term2/capstone/arvato_data/Udacity_AZDIAS_052018.csv',

        azdias.head()
```

```
In [ ]: 'Shape', azdias.shape
```

```
In [ ]: azdias.info()
```

```
In [ ]: azdias.describe()
```

Examine customers dataframe

```
In [ ]: # load in the data
        customers = pd.read_csv('../../data/Term2/capstone/arvato_data/Udacity_CUSTOMERS_052018.

        customers.head()
```

```
In [ ]: 'Shape', customers.shape
```

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

```
In [ ]: customers.describe()
```

Further consideration of datasets, categorical variables

```
In [ ]: # Look at extra columns

        extra = ['CUSTOMER_GROUP', 'ONLINE_PURCHASE', 'PRODUCT_GROUP']
        customers['ONLINE_PURCHASE'] = customers['ONLINE_PURCHASE'].astype(str)
        customers['ONLINE_PURCHASE'].replace({'0': 'No', '1': 'Yes'}, inplace=True)

        print(customers[extra].dtypes)

        fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(21,7))
        for i in range(3):
            customers[extra[i]].value_counts().plot(kind='bar', ax=ax[i], width=.9)
            ax[i].legend(loc='best')
        ax[1].set_title('Groups of customers, purchases, and products', fontsize=15, y=1.03)
        plt.show()
```

```
In [ ]: # Save this additional info in another customers_extra DF

        extra = ['CUSTOMER_GROUP', 'ONLINE_PURCHASE', 'PRODUCT_GROUP']
        customers_extra = customers[extra]
        customers.drop(extra, axis=1, inplace=True)
```

```
In [ ]: cat_cols_azdias = azdias.select_dtypes(include='object').columns
        cat_cols_customers = customers.select_dtypes(include='object').columns
        set(cat_cols_customers).difference(set(cat_cols_azdias))
```

```
In [ ]: # Plot up to top-20

        plt.rc('xtick', labelsize=12)

        for column in cat_cols_azdias:
            if column != 'EINGEFUEGT_AM':
                fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(21,7))
                azdias[column].value_counts().iloc[:20].plot(kind='bar', ax=ax[0], width=0.9, cm
                customers[column].value_counts().iloc[:20].plot(kind='bar', ax=ax[1], width=0.9,
                plt.suptitle(column, fontsize=15)
                plt.show()
```

```
In [ ]: print(azdias['EINGEFUEGT_AM'].dtype)
        print(azdias['EINGEFUEGT_AM'].unique())
        azdias['EINGEFUEGT_AM'].value_counts().describe()
```

```
In [ ]: print(customers['EINGEFUEGT_AM'].dtype)
        print(customers['EINGEFUEGT_AM'].unique())
        customers['EINGEFUEGT_AM'].value_counts().describe()
```

Further consideration of datasets, distribution of numerical variables As my project reviewer suggested, I examined the distribution of the corresponding column values in both AZDIAS and CUSTOMERS datasets. As a result of manual comparing the numbers on figures, a list was created with columns in which distributions aren't similar. In further analysis, these columns will be retained from removing columns with multiple missing values. It's my interpretation that if the same columns from both DataFrames have different distributions, we can use this additional information in our clustering model.

```
In [ ]: columns_ = azdias.select_dtypes(include='int').columns.tolist()

        # All figures gets a lot of space, so only one example "D19_KONSUMTYP_MAX" column is plo
        fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(21,7))
        azdias['D19_KONSUMTYP_MAX'].plot.hist(ax=ax[0], cmap='GnBu_r')
        customers['D19_KONSUMTYP_MAX'].plot.hist(ax=ax[1], cmap='GnBu_r')
        plt.suptitle('D19_KONSUMTYP_MAX', fontsize=15)
        plt.show()

        '''
        # Plot columns whichdtype is integer
        for column in columns_:
            fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(21,7))
            azdias[column].plot.hist(ax=ax[0], cmap='GnBu_r')
            customers[column].plot.hist(ax=ax[1], cmap='GnBu_r')
            plt.suptitle(column, fontsize=15)
            plt.show()'''
```

```
In [ ]: columns_ = azdias.select_dtypes(include='float').columns.tolist()

        # All figures gets a lot of space, so only one example "D19_SOZIALES" column is plotted
        fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(21,7))
        azdias['D19_SOZIALES'].plot.hist(ax=ax[0], cmap='GnBu_r')
        customers['D19_SOZIALES'].plot.hist(ax=ax[1], cmap='GnBu_r')
        plt.suptitle('D19_SOZIALES', fontsize=15)
        plt.show()

        '''
        # Plot columns whichdtype is float
        for column in columns_:
            fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(21,7))
            azdias[column].plot.hist(ax=ax[0], cmap='GnBu_r')
            customers[column].plot.hist(ax=ax[1], cmap='GnBu_r')
            plt.suptitle(column, fontsize=15)
            plt.show()'''

In [ ]: cols_to_keep = ['D19_KONSUMTYP_MAX', 'FINANZ_ANLEGER', 'FINANZ_MINIMALIST', 'FINANZTYP',
                        'PRAEGENDE_JUGENDJAHRE', 'SEMIO_MAT', 'SEMIO_VERT', 'SHOPPER_TYP', 'ZABE
                        'AKT_DAT_KL', 'ALTER_HH', 'CJT_GESAMTTYP', 'CJT_TYP_1', 'CJT_TYP_2', 'D1
                        'EINGEZOGENAM_HH_JAHR', 'GFK_URLAUBERTYP', 'HH_EINKOMMEN_SCORE', 'KBA05_
                        'KBA05_BAUMAX', 'KBA05_CCM4', 'KBA05_GBZ', 'KBA05_MAXAH', 'KBA05_MOD1',
                        'LP_STATUS_FEIN', 'LP_STATUS_GROB', 'MOBI_REGIO', 'ORTSGR_KLS9', 'RT_KEI
                        'VHA', 'VK_DHT4A', 'VK_DISTANZ', 'VK_ZG11']

        for column in cols_to_keep:
            fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(21,7))
            azdias[column].plot.hist(ax=ax[0], cmap='GnBu_r')
            customers[column].plot.hist(ax=ax[1], cmap='GnBu_r')
            plt.suptitle(column, fontsize=15)
            plt.show()
```

Work with DIAS Information Levels and Values / 2017

```
In [ ]: # load in a top-level list of attributes and descriptions, organized by informational ca

        levels_df = pd.read_excel('DIAS Information Levels - Attributes 2017.xlsx', skiprows=1)
        levels_df = levels_df[['Information level', 'Attribute', 'Description']]
        levels_df.loc[0, 'Information level'] = 'Person'
        levels_df.loc[88:96, 'Information level'] = 'Microcell (RR3_ID)'
        levels_df['Information level'] = levels_df['Information level'].fillna(method='ffill')
        levels_df.head()

In [ ]: levels_df['Information level'].unique()

In [ ]: # load in a detailed mapping of data values for each feature in alphabetical order
```

```
values_df = pd.read_excel('DIAS Attributes - Values 2017.xlsx', skiprows=1)
values_df.drop('Unnamed: 0', axis=1, inplace=True)
values_df = values_df.fillna(method='ffill')
values_df.head()
```

```
In [ ]: details = values_df.merge(levels_df, on=['Attribute', 'Description'], how='left')
details.head()
```

```
In [ ]: details.isnull().sum()
```

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

```
In [ ]: # Number of all values and unique Attributes in DIAS data
details.shape, details.Attribute.unique().shape
```

```
In [ ]: details.Meaning.value_counts().iloc[:50].plot.bar(figsize=(21,7), width=.9)
plt.title('Distribution of DIAS Attributes Values', fontsize=15)
plt.show()
```

In the following part such values as "unknown" or "unknown / no main age detectable", etc. will be replaced with NaNs.

Replace unknown values in azdias and customers with NaN using details DF

```
In [ ]: # Get list of Attribute - Value where "unknown" or "no transaction(s) known" is present

details['meaning'] = details.Meaning.str.lower().astype(str)
unknown = details[(details.meaning.str.contains('unknown')) |
                  (details.meaning.isin(['no transactions known', 'no transaction known'
unknown.set_index('Attribute', inplace=True)
unknown.head()
```

```
In [ ]: def replace_unknown_with_nan(df, unknown):

    cols_df = df.columns
    for column in unknown.index:
        if column in cols_df:
            col_values = df[column].unique().tolist()
            unknown_vals = unknown.loc[column]['Value']
            for val in col_values:
                if isinstance(unknown_vals, int):
                    if val == unknown_vals:
                        df[column] = df[column].replace(val, np.nan)
                else:
                    if str(val) in unknown_vals.split():
                        df[column] = df[column].replace(val, np.nan)
    return df
```

Apply function replace_unknown_with_nan to azdias dataframe

```
In [ ]: azdias.isnull().sum().sort_values(ascending=False).iloc[:40].plot.bar(figsize=(21,7), wi
        plt.title('Missing values before editting', fontsize=15)
        plt.show()

In [ ]: azdias = replace_unknown_with_nan(azdias, unknown)

In [ ]: azdias.isnull().sum().sort_values(ascending=False).iloc[:40].plot.bar(figsize=(21,7), wi
        plt.title('Missing values after editting', fontsize=15)
        plt.show()
```

Apply function replace_unknown_with_nan to customers dataframe

```
In [ ]: customers.isnull().sum().sort_values(ascending=False).iloc[:50].plot.bar(figsize=(21,7),
        plt.title('Missing values before editting', fontsize=15)
        plt.show()

In [ ]: customers = replace_unknown_with_nan(customers, unknown)

In [ ]: customers.isnull().sum().sort_values(ascending=False).iloc[:40].plot.bar(figsize=(21,7),
        plt.title('Missing values after editting', fontsize=15)
        plt.show()
```

Drop columns from azdias and customers where there's no corresponding value in Attribute in details DF

```
In [ ]: # Differences and similarities in azdias and details DF

        common_cols = set(azdias.columns.tolist()).intersection(set(details.Attribute.unique().t
        different_cols = set(azdias.columns.tolist()).symmetric_difference(set(details.Attribute
        different_azdias_cols = set(azdias.columns.tolist()).difference(set(details.Attribute.un
        different_details_cols = set(details.Attribute.tolist()).difference(set(azdias.columns.u

        print('Same for both azdias and details', len(common_cols))
        print('Different for both azdias and details', len(different_cols))
        print('Different for azdias', len(different_azdias_cols))
        print('Different for details', len(different_details_cols))

In [ ]: # Differences and similarities in customers and details DF

        common_cols = set(customers.columns.tolist()).intersection(set(details.Attribute.unique(
        different_cols = set(customers.columns.tolist()).symmetric_difference(set(details.Attrib
        different_customers_cols = set(customers.columns.tolist()).difference(set(details.Attrib
        different_details_cols = set(details.Attribute.tolist()).difference(set(customers.column

        print('Same for both customers and details', len(common_cols))
        print('Different for both customers and details', len(different_cols))
        print('Different for customers', len(different_azdias_cols))
        print('Different for details', len(different_details_cols))
```

It seems that D19 columns with "RZ" in the end of column names in DIAS Attributes DF ("RZ" stands for "Herzogtum Lauenburg" region in Germany) means the same as the corresponding columns of total population. So, the unknown or missing values could also be replaced with NaN. The same could be applied to columns CAMEO_INTL_2015 (azdias DF) and CAMEO_DEUINTL_2015 () are the same (and probably "DEU" means "Deutsche Eislauf-Union", but it's not exact).

```
In [ ]: details[details.Attribute == 'D19_BANKEN_GROSS_RZ'].head()
```

```
In [ ]: details.Attribute = details.Attribute.str.replace('_RZ', '')
        details[details.Attribute == 'D19_BANKEN_GROSS'].head()
```

```
In [ ]: azdias = replace_unknown_with_nan(azdias, unknown)

        azdias.isnull().sum().sort_values(ascending=False).iloc[:50].plot.bar(figsize=(21,7), wi
        plt.title('Missing values after editting', fontsize=15)
        plt.show()
```

```
In [ ]: customers = replace_unknown_with_nan(customers, unknown)

        customers.isnull().sum().sort_values(ascending=False).iloc[:50].plot.bar(figsize=(21,7),
        plt.title('Missing values after editting', fontsize=15)
        plt.show()
```

Convert categorical columns to numeric where possible

```
In [ ]: def object_toNumeric(df):

            cat_cols = df.select_dtypes(include='object').columns
            for column in cat_cols:
                print(column)
                unqiue_vals = df[column].unique()
                print('Unique values in', column, unqiue_vals, '\n')
                if ('X' in unqiue_vals) or ('XX' in unqiue_vals):
                    df[column] = df[column].replace({'X': np.nan, 'XX': np.nan})
                    df[column] = pd.to_numeric(df[column], errors='coerse')

            return df
```

```
In [ ]: azdias = object_toNumeric(azdias)
        azdias.select_dtypes(include='object').head()
```

```
In [ ]: customers = object_toNumeric(customers)
        customers.select_dtypes(include='object').head()
```

Convert categorical columns to datetime where possible

```
In [ ]: def object_toDatetime(df, column):
```

```
            df[column] = pd.to_datetime(df[column])
            df['YEAR'] = df[column].dt.year
            df['MONTH'] = df[column].dt.month
            df.drop(column, axis=1, inplace=True)

            return df
```

```
In [ ]: azdias = object_toDatetime(azdias, 'EINGEFUEGT_AM')
        azdias[['YEAR', 'MONTH']].head()
```

```
In [ ]: customers = object_toDatetime(customers, 'EINGEFUEGT_AM')
        customers[['YEAR', 'MONTH']].head()
```

Missing values

```
In [ ]: nan_azdias = azdias.isnull().sum()
        nan_azdias = nan_azdias.sort_values(ascending=False)
        nan_azdias_cols = nan_azdias[nan_azdias > 0]
        print('Number of columns with NaNs is', len(nan_azdias_cols))
        proportion_azdias = nan_azdias_cols / azdias.shape[0]
        print(proportion_azdias[proportion_azdias > .9])
```

```
In [ ]: nan_customers = customers.isnull().sum()
        nan_customers = nan_customers.sort_values(ascending=False)
        nan_customers_cols = nan_customers[nan_customers > 0]
        print('Number of columns with NaNs is', len(nan_customers_cols))
        proportion_customers = nan_customers_cols / customers.shape[0]
        print(proportion_customers[proportion_customers > .9])
```

The calculated statistics for each feature in demographic data

```
In [ ]: def build_feat_info(df):

            return pd.DataFrame({
                'value_count': [df[column].count() for column in df.columns],
                'value_distinct' : [df[column].unique().shape[0] for column in df.columns],
                'num_nans':  [df[column].isnull().sum() for column in df.columns],
                'percent_nans' : [round(df[column].isnull().sum()/df[column].shape[0], 3) for co
            },  index = df.columns)

        feat_info_azdias =  build_feat_info(azdias)
        feat_info_azdias.sort_values(by=['percent_nans'], ascending=False)
```

```
In [ ]: feat_info_customers =  build_feat_info(customers)
        feat_info_customers.sort_values(by=['percent_nans'], ascending=False)
```

```
In [ ]: # Delete columns with more than 90% missing values

        cols_to_drop = nan_azdias_cols[nan_azdias_cols / azdias.shape[0] > .9].index.tolist()
        cols_to_drop
```

```
In [ ]: azdias.drop(cols_to_drop, axis=1, inplace=True)

In [ ]: cols_to_drop = nan_customers_cols[nan_customers_cols / customers.shape[0] > .9].index.to
        cols_to_drop

In [ ]: customers.drop(cols_to_drop, axis=1, inplace=True)

In [ ]: # Column which has a little bit less missing values in azdias than in customers DF

        difference = (set(proportion_customers[proportion_customers > .9].index)
                          .difference(set(proportion_azdias[proportion_azdias > .9].index)))
        print(difference)
        print(proportion_azdias[difference])

In [ ]: azdias.drop(difference, axis=1, inplace=True)

In [ ]: # Visualize distribution of missing values in top-30 columns in azdias

        nan_azdias = azdias.isnull().sum()
        nan_azdias = nan_azdias.sort_values(ascending=False)
        nan_azdias_cols = nan_azdias[nan_azdias > 0]

        nan_azdias_cols.iloc[:30].plot(kind='bar', figsize=(21,7), width=.9, color='green')
        plt.xticks(rotation=90)
        plt.title('Distribution of NaNs in azdias DataFrame', fontsize=15)
        plt.xlabel('Top-30 columns with NaNs')
        plt.ylabel('Number of NaNs')

        plt.show()

In [ ]: # Visualize distribution of missing values in top-30 columns in customers

        nan_customers = customers.isnull().sum()
        nan_customers = nan_customers.sort_values(ascending=False)
        nan_customers_cols = nan_customers[nan_customers > 0]

        nan_customers_cols.iloc[:30].plot(kind='bar', figsize=(21,7), width=.9, color='green')
        plt.xticks(rotation=90)
        plt.title('Distribution of NaNs in customers DataFrame', fontsize=15)
        plt.xlabel('Top-30 columns with NaNs')
        plt.ylabel('Number of NaNs')

        plt.show()

In [ ]: print('Shape after deleting columns with more than 90% missing values')
        print('Azdias', azdias.shape)
        print('Customers', customers.shape)
```

Drop columns and create new ones

```
In [ ]: # Create column which represent sum of NaN in each row

        def create_num_missing_column(df):
            df['Num_missing'] = df.isnull().sum(axis=1)

            return df

In [ ]: azdias = create_num_missing_column(azdias)

In [ ]: customers = create_num_missing_column(customers)

In [ ]: # For columns with NaN more than 10% and less 90% replace missing values with 0s, and no

        def tranform_cols_with_missing(df, cols_to_keep):
            nan_df = df.isnull().sum() / df.shape[0]
            nan_cols = nan_df[(nan_df > .1) & (nan_df < .9)]
            for column in nan_cols.index:
                if column not in cols_to_keep:
                    df[column] = np.where(df[column].isnull(), 0, 1)

            return df

In [ ]: azdias = tranform_cols_with_missing(azdias, cols_to_keep)

In [ ]: customers = tranform_cols_with_missing(customers, cols_to_keep)

In [ ]: # For columns with NaN in cols_to_keep

        def missing_vals_in_cols_to_keep(df, cols_to_keep):
            nan_df = df.isnull().sum() / df.shape[0]
            nan_cols = nan_df[(nan_df > .1) & (nan_df < .9)]
            for column in df[nan_cols.index]:
                df[column] = df[column].replace(np.nan, -1)

            return df

In [ ]: azdias = missing_vals_in_cols_to_keep(azdias, cols_to_keep)

In [ ]: customers = missing_vals_in_cols_to_keep(customers, cols_to_keep)

In [ ]: # Work with azdias DF where # of missing values < 10%

        nan_azdias = azdias.isnull().sum() / azdias.shape[0]
        nan_cols = nan_azdias[(nan_azdias > 0) & (nan_azdias < .1)]
        print(nan_cols)
```

Replace missing values with modes of correspondng columns.

```
In [ ]: for column in azdias[nan_cols.index]:
            col_mode = azdias[column].mode().values[0]
            azdias[column] = azdias[column].replace(np.nan, col_mode)

        # azdias.isnull().sum().sum()

In [ ]: # Work with customers DF where # of missing values < 10%

        nan_customers = customers.isnull().sum() / customers.shape[0]
        nan_cols = nan_customers[(nan_customers > 0) & (nan_customers < .1)]
        print(nan_cols)

In [ ]: customers[nan_cols.index].plot.hist(figsize=(21,7), alpha=.5, bins=20)
        plt.legend(loc='best')
        plt.title('Histograms of columns with <10% missing values', fontsize=15)
        plt.show()
```

In this case less than 2% of answers are missing. Since, values in this columns are discrete, and distributions are more or less normal (except "CJT_TYP_1" column), missing values will be replaced with modes of correspondng columns.

```
In [ ]: for column in customers[nan_cols.index]:
            col_mode = customers[column].mode().values[0]
            customers[column] = customers[column].replace(np.nan, col_mode)

        # customers.isnull().sum().sum()

In [ ]: print('# of missing in azdias', azdias.isnull().sum().sum())
        print('# of missing in customers', customers.isnull().sum().sum())

In [ ]: print('Whether all values in LNR column are unique in azdias DF -', len(azdias.LNR.uniqu
        print('Whether all values in LNR column are unique in customers DF - ',
              len(customers.LNR.unique()) == customers.shape[0])

        print('# of common values in LNR column', len(set(customers.LNR).intersection(set(azdias
        print('# of different values in LNR column', len(set(customers.LNR).difference(set(azdia

        # So, we can drop LNR column in both datasets
        azdias.drop('LNR', axis=1, inplace=True)
        customers.drop('LNR', axis=1, inplace=True)
```

Create dummies from categorical variables

```
In [ ]: def create_dummies(df):

            cat_cols = df.select_dtypes(include='object').columns
            for column in cat_cols:
                num_unique = len(df[column].value_counts())
                if (num_unique == 2) or (num_unique == 3 and np.nan in df[column].value_counts()
```

```
                values = df[column].value_counts()
                df[column] = df[column].replace({values.index[0]: 0, values.index[1]: 1})
            else:
                df = pd.concat([df.drop(column, axis=1), pd.get_dummies(df[column], drop_fir

        return df
```

```
In [ ]: azdias = create_dummies(azdias)
        azdias.dtypes.value_counts()
```

```
In [ ]: azdias.dtypes.value_counts().plot.bar()
        plt.show()
```

```
In [ ]: customers = create_dummies(customers)
        customers.dtypes.value_counts()
```

```
In [ ]: customers.dtypes.value_counts().plot.bar()
        plt.show()
```

```
In [ ]: azdias.shape, customers.shape, customers_extra.shape
```

Function with pre-processing steps to clean all of the datasets before work with them.

```
In [ ]: def clean_demographic_data(df, unknown, cat_column='EINGEFUEGT_AM', cols_to_keep=cols_to

            ##############################
            # The calculated statistics for each feature in demographic data
            feat_info_df =  build_feat_info(df)
            print(feat_info_df.sort_values(by=['percent_nans'], ascending=False))

            ##############################
            # Replace unknown values with NaNs

            cols_df = df.columns
            for column in unknown.index:
                if column in cols_df:
                    col_values = df[column].unique().tolist()
                    unknown_vals = unknown.loc[column]['Value']
                    for val in col_values:
                        if isinstance(unknown_vals, int):
                            if val == unknown_vals:
                                df[column] = df[column].replace(val, np.nan)
                        else:
                            if str(val) in unknown_vals.split():
                                df[column] = df[column].replace(val, np.nan)


            ##############################
            # Convert categorical columns to numeric where possible
```

```
cat_cols = df.select_dtypes(include='object').columns
for column in cat_cols:
    unqiue_vals = df[column].unique()
    if ('X' in unqiue_vals) or ('XX' in unqiue_vals):
        df[column] = df[column].replace({'X': np.nan, 'XX': np.nan})
        df[column] = pd.to_numeric(df[column], errors='coerse')

##############################
# Convert categorical columns to datetime where possible

df[cat_column] = pd.to_datetime(df[cat_column])
df['YEAR'] = df[cat_column].dt.year
df['MONTH'] = df[cat_column].dt.month
df.drop(cat_column, axis=1, inplace=True)

##############################
# Missing values

nan_df = df.isnull().sum()
nan_df = nan_df.sort_values(ascending=False)
nan_df_cols = nan_df[nan_df > 0]
proportion_df = nan_df_cols / df.shape[0]

# Delete columns with more than 90% missing values
cols_to_drop = nan_df_cols[nan_df_cols / df.shape[0] > .9].index.tolist()
df.drop(cols_to_drop, axis=1, inplace=True)

##############################
# Create column which represent sum of NaN in each row
df['Num_missing'] = df.isnull().sum(axis=1)

##############################
# For columns with NaN more than 10% and less 90% replace missing values with 0s, an
# (except cols_to_keep)

nan_df = df.isnull().sum() / df.shape[0]
nan_cols = nan_df[(nan_df > .1) & (nan_df < .9)]
for column in nan_cols.index:
    if column not in cols_to_keep:
        print(column)
        df[column] = np.where(df[column].isnull(), 0, 1)

##############################
# For columns with NaN in cols_to_keep

nan_df = df.isnull().sum() / df.shape[0]
nan_cols = nan_df[(nan_df > .1) & (nan_df < .9)]
```

```
    for column in df[nan_cols.index]:
        df[column] = df[column].replace(np.nan, -1)

    ##############################
    # LNR column
    print('Whether all values in LNR column are unique in azdias DF -', len(df.LNR.uniqu
    df.drop('LNR', axis=1, inplace=True)

    ##############################
    # Work with customers DF where # of missing values < 10%

    nan_df = df.isnull().sum() / df.shape[0]
    nan_cols = nan_df[(nan_df > 0) & (nan_df < .1)]

    for column in df[nan_cols.index]:
        col_mode = df[column].mode().values[0]
        df[column] = df[column].replace(np.nan, col_mode)

    ##############################
    # Create dummies from categorical variables

    cat_cols = df.select_dtypes(include='object').columns
    for column in cat_cols:
        num_unique = len(df[column].value_counts())
        if (num_unique == 2) or (num_unique == 3 and np.nan in df[column].value_counts()
            values = df[column].value_counts()
            df[column] = df[column].replace({values.index[0]: 0, values.index[1]: 1})
        else:
            df = pd.concat([df.drop(column, axis=1), pd.get_dummies(df[column], drop_fir



    print('Final check of missing values', df.isnull().sum().sum())

    return df
```

In [ ]:

In [ ]:

1.2 Part 1: Customer Segmentation Report

The main bulk of your analysis will come in this part of the project. Here, you should use unsupervised learning techniques to describe the relationship between the demographics of the company's existing customers and the general population of Germany. By the end of this part, you should be able to describe parts of the general population that are more likely to be part of the mail-order company's main customer base, and which parts of the general population are less so.

Standardize values

```
In [ ]: azdias_cols = azdias.columns
        customers_cols = customers.columns

        scaler = StandardScaler()
        azdias_scaled = scaler.fit_transform(azdias)
        print('azdias done!')
        customers_scaled = scaler.transform(customers)
```

```
In [ ]: pd.DataFrame(azdias_scaled).head()
```

Implement PCA

```
In [ ]: pca = PCA(.9)
        azdias_reduced = pca.fit_transform(azdias_scaled)
        print('azdias done!')
        customers_reduced = pca.transform(customers_scaled)
        print('Explained variance ratio of azdias DF', pca.explained_variance_ratio_)
```

```
In [ ]: print('New shape of azdias', azdias_reduced.shape)
        print('New shape of customers', customers_reduced.shape)
```

```
In [ ]: plt.subplots(figsize=(21,7))
        plt.plot(pca.explained_variance_ratio_.cumsum())
        plt.title('Cumulative explained variance')
        plt.xlabel('Number of components')
        plt.ylabel('Explained Variance')
        plt.show()
```

KMeans

```
In [ ]: # Choose an optimal # of clusters
        # With help of "How to determine the optimal number of clusters for k-means clustering"
        # https://blog.cambridgespark.com/how-to-determine-the-optimal-number-of-clusters-for-k-

        sum_of_squared_distances = []
        for k in range(2,31):
            print('Fit {} clusters'.format(k))
            kmeans = KMeans(n_clusters=k)
            kmeans = kmeans.fit(azdias_reduced)
            sum_of_squared_distances.append(kmeans.inertia_) # SSD of samples to their closest c
```

```
In [ ]: plt.subplots(figsize=(21,7))
        plt.plot(range(2,31), sum_of_squared_distances)
        plt.xlabel('Number of clusters, k')
        plt.ylabel('Sum of squared distances')
        plt.title('Elbow Method For Optimal k, SSE', fontsize=15)
        plt.show()
```

It's very difficult to determine the optimal # of clusters from this figure. So, the further analysis of 6 and 14 clusters will be performed.

```
In [ ]: def kmeans(df, n_clusters):
            df = pd.DataFrame(df)

            kmeans = KMeans(n_clusters=n_clusters).fit(df)

            cluster_map = pd.DataFrame()
            cluster_map['cluster'] = kmeans.labels_

            return kmeans, cluster_map
```

k=6

```
In [ ]: kmeans_azdias_6, cluster_map_6 = kmeans(azdias_reduced, 6)

In [ ]: preds_azdias_6 = pd.DataFrame(kmeans_azdias_6.predict(azdias_reduced), columns=['AZDIAS'
        preds_customers_6 = pd.DataFrame(kmeans_azdias_6.predict(customers_reduced), columns=['C

        preds_6 = pd.concat([preds_azdias_6['AZDIAS'].value_counts(), preds_customers_6['CUSTOME
                                                                axis=1, sort=False)

        preds_6

In [ ]: preds_azdias_6.shape, preds_customers_6.shape

In [ ]: preds.plot(kind='bar', figsize=(21,7), width=.9)
        plt.title('Clusters of general population of Germany vs. customers of a mail-order compa
        plt.legend()
        plt.show()
```

k=14

```
In [ ]: kmeans_azdias_14, cluster_map_14 = kmeans(azdias_reduced, 14)

In [ ]: preds_azdias_14 = pd.DataFrame(kmeans_azdias_14.predict(azdias_reduced), columns=['AZDIA
        preds_customers_14 = pd.DataFrame(kmeans_azdias_14.predict(customers_reduced), columns=[

        preds_14 = pd.concat([preds_azdias_14['AZDIAS'].value_counts(), preds_customers_14['CUST
                                                                axis=1, sort=False)

        preds_14

In [ ]: preds_azdias_14.shape, preds_customers_14.shape

In [ ]: preds.plot(kind='bar', figsize=(21,7), width=.9)
        plt.title('Clusters of general population of Germany vs. customers of a mail-order compa
        plt.legend()
        plt.show()
```

Discuss results For the prepared demographic data of the general population of Germany and customers, the use of 6 as well as 14 clusters leads to the fact that a disproportion appears in the results, and some of the people from the customers simply do not fall into the corresponding clusters of the entire population.

However, the most important factor is business request. It include an amount of resources the company would spend, target audience (for example, online purchases of only one product of cosmetics, etc.). In this analysis I use results of segmentation into 6 and 14 clusters, but it can still be any number of clusters. Thus, significantly reducing costs and not covering all customers. And of course, further research needs to look at the results of hierarchical clustering in order to compare several clustering methods. This analysis will take more resources, including time, than KMeans clustering.

```
In [ ]: preds_6['AZDIAS'] = round(preds_6['AZDIAS'] / preds_6['AZDIAS'].sum() * 100, 1)
        preds_6['CUSTOMERS'] = round(preds_6['CUSTOMERS'] / preds_6['CUSTOMERS'].sum() * 100, 1)
        preds_6
```

```
In [ ]: preds_14['AZDIAS'] = round(preds_14['AZDIAS'] / preds_14['AZDIAS'].sum() * 100, 1)
        preds_14['CUSTOMERS'] = round(preds_14['CUSTOMERS'] / preds_14['CUSTOMERS'].sum() * 100,
        preds_14
```

```
In [ ]: 'Shape extra customers DF' + str(customers_extra.shape)
```

Analyze 6 clusters scenario

```
In [ ]: preds_customers_6 = pd.concat([preds_customers_6, customers_extra], axis=1, sort=False)
        print('Shape (6 clusters)', preds_customers_6.shape)
        preds_customers_6.head()
```

```
In [ ]: round(pd.crosstab(preds_customers_6['CUSTOMERS'], preds_customers_6['CUSTOMER_GROUP'])
              / preds_customers_6.shape[0] * 100, 1)
```

```
In [ ]: round(pd.crosstab(preds_customers_6['CUSTOMERS'], preds_customers_6['ONLINE_PURCHASE'])
              / preds_customers_6.shape[0] * 100, 1)
```

```
In [ ]: round(pd.crosstab(preds_customers_6['CUSTOMERS'], preds_customers_6['PRODUCT_GROUP'])
              / preds_customers_6.shape[0] * 100, 1)
```

```
In [ ]: preds_customers_6['ONLINE_PURCHASE'] = preds_customers_6['ONLINE_PURCHASE'].replace({'Ye
        round(pd.pivot_table(data=preds_customers_6, index='CUSTOMERS', columns='CUSTOMER_GROUP'
                             aggfunc='mean') * 100, 1)
```

```
In [ ]: round(pd.pivot_table(data=preds_customers_6, index='CUSTOMERS', columns='PRODUCT_GROUP',
                             aggfunc='mean') * 100, 1)
```

Analyze 14 clusters scenario

```
In [ ]: preds_customers_14 = pd.concat([preds_customers_14, customers_extra], axis=1, sort=False
        print('Shape (14 clusters)', preds_customers_14.shape)
        preds_customers_14.head()
```

```
In [ ]: round(pd.crosstab(preds_customers_14['CUSTOMERS'], preds_customers_14['CUSTOMER_GROUP'])
              / preds_customers_14.shape[0] * 100, 1)

In [ ]: round(pd.crosstab(preds_customers_14['CUSTOMERS'], preds_customers_14['ONLINE_PURCHASE']
              / preds_customers_14.shape[0] * 100, 1)

In [ ]: round(pd.crosstab(preds_customers_14['CUSTOMERS'], preds_customers_14['PRODUCT_GROUP'])
              / preds_customers_14.shape[0] * 100, 1)

In [ ]: preds_customers_14['ONLINE_PURCHASE'] = preds_customers_14['ONLINE_PURCHASE'].replace({'
        round(pd.pivot_table(data=preds_customers_14, index='CUSTOMERS', columns='CUSTOMER_GROUP
                             aggfunc='mean') * 100, 1)

In [ ]: round(pd.pivot_table(data=preds_customers_14, index='CUSTOMERS', columns='PRODUCT_GROUP'
                             aggfunc='mean') * 100, 1)

In [ ]:

In [ ]:
```

1.3 Part 2: Supervised Learning Model

Now that you've found which parts of the population are more likely to be customers of the mail-order company, it's time to build a prediction model. Each of the rows in the "MAILOUT" data files represents an individual that was targeted for a mailout campaign. Ideally, we should be able to use the demographic information from each individual to decide whether or not it will be worth it to include that person in the campaign.

The "MAILOUT" data has been split into two approximately equal parts, each with almost 43 000 data rows. In this part, you can verify your model with the "TRAIN" partition, which includes a column, "RESPONSE", that states whether or not a person became a customer of the company following the campaign. In the next part, you'll need to create predictions on the "TEST" partition, where the "RESPONSE" column has been withheld.

```
In [ ]: mailout_train = pd.read_csv('../../data/Term2/capstone/arvato_data/Udacity_MAILOUT_05201
        mailout_train.head()

In [ ]: 'Shape', mailout_train.shape

In [ ]: y = mailout_train['RESPONSE']
        X = mailout_train.drop('RESPONSE', axis=1)
        X = clean_demographic_data(X, unknown)

In [ ]: 'Shape X', X.shape

In [ ]: plt.rc('xtick', labelsize=12)

        plt.subplots(figsize=(21,7))
        y.value_counts().iloc[:20].plot(kind='bar', width=0.9, cmap='GnBu_r')
        plt.title('Response', fontsize=15)
        plt.show()
```

```
In [ ]: # Split data into train and test sets
        X_train, X_test, y_train, y_test = train_test_split(X, y)

In [ ]: X_train.shape, X_test.shape, y_train.shape, y_test.shape

In [ ]: pipeline = Pipeline([
                ('scale', scaler),
                ('reduce_dim', pca),
                ('clf', RandomForestClassifier(class_weight='balanced'))
            ])

In [ ]: # Train pipeline
        pipeline.fit(X_train, y_train)

In [ ]: pred = pipeline.predict(X_test)
        print(pred.shape, y_test.shape)

        precision_ = precision_score(y_test, pred)
        recall_ = recall_score(y_test, pred)
        f1_ = f1_score(y_test, pred)
        accuracy_ = accuracy_score(y_test, pred)

        precision_, recall_, f1_, accuracy_

In [ ]: y_train.mean(), y_test.mean(), pred.mean()

In [ ]: # Use grid search to find better parameters

        parameters = {
                'clf__n_estimators': [50, 100, 200],
                'criterion': ['gini', 'entropy'],
                'max_features': ['auto', 'sqrt', 'log2', None],
                'bootstrap': [True, False]
            }

        cv = GridSearchCV(pipeline, param_grid=parameters)

In [ ]: cv.fit(X_train, y_train)

In [ ]: pred_2 = cv.predict(X_test)
        print(pred_2.shape, y_test.shape)

        precision_ = precision_score(y_test, pred_2)
        recall_ = recall_score(y_test, pred_2)
        f1_ = f1_score(y_test, pred_2)
        accuracy_ = accuracy_score(y_test, pred_2)

        precision_, recall_, f1_, accuracy_

In [ ]:

In [ ]:
```

1.4 Part 3: Kaggle Competition

Now that you've created a model to predict which individuals are most likely to respond to a mailout campaign, it's time to test that model in competition through Kaggle. If you click on the link here, you'll be taken to the competition page where, if you have a Kaggle account, you can enter. If you're one of the top performers, you may have the chance to be contacted by a hiring manager from Arvato or Bertelsmann for an interview!

Your entry to the competition should be a CSV file with two columns. The first column should be a copy of "LNR", which acts as an ID number for each individual in the "TEST" partition. The second column, "RESPONSE", should be some measure of how likely each individual became a customer – this might not be a straightforward probability. As you should have found in Part 2, there is a large output class imbalance, where most individuals did not respond to the mailout. Thus, predicting individual classes and using accuracy does not seem to be an appropriate performance evaluation method. Instead, the competition will be using AUC to evaluate performance. The exact values of the "RESPONSE" column do not matter as much: only that the higher values try to capture as many of the actual customers as possible, early in the ROC curve sweep.

```
In [ ]: mailout_test = pd.read_csv('../../data/Term2/capstone/arvato_data/Udacity_MAILOUT_052018
        mailout_test.head()

In [ ]: mailout_test.shape

In [ ]: lnr = mailout_test['LNR']
        mailout_test_cleaned = clean_demographic_data(mailout_test, unknown)

In [ ]: pred_3 = pipeline.predict(mailout_test_cleaned)
        pred_4 = cv.predict(mailout_test_cleaned)

In [ ]: arvato_capstone_submission_3 = pd.DataFrame({'LNR': lnr, 'RESPONSE': pred_3})
        arvato_capstone_submission_4 = pd.DataFrame({'LNR': lnr, 'RESPONSE': pred_4})

In [ ]: arvato_capstone_submission_3.to_csv('Arvato_Capstone_Submission_3.csv', index=False)
        arvato_capstone_submission_4.to_csv('Arvato_Capstone_Submission_4.csv', index=False)

In [ ]:

In [ ]:

In [ ]:
```