Machine Learning Engineer Nanodegree

Capstone Project
Youssof Yassin
January 29th, 2021

I. Definition

Project Overview

Arvato is one of the many divisions that make up Bertelmann, a media, services and education group. It is headquartered in Gütersloh, Germany and deals in customer reports, information technology, logistics and finance. Since Arvato is an international services company and has been around for many years, it has acquired a significant amount of data, which it means to make use of as much as possible. With the expanding release of data science and machine learning tools nowadays, it has become the norm for every major company to start using data-related techniques to target customers, instead of just pure intuition.

The datasets are provided primarily by Arvato company to Udacity students for this specific type of capstone project. Datasets used would be:

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).

Demographics data, that was for individuals who were targets of a marketing campaign, was split into 2 datasets:

Udacity_MAILOUT_052018_TRAIN.csv: Contains the data we will be training our algorithms on; 42 982 persons (rows) x 367 (columns).
Udacity_MAILOUT_052018_TEST.csv: Contains the data we will be trying to predict; 42 833 persons (rows) x 366 (columns).

The **CUSTOMERS** dataset will be used to compare with the general population **AZDIAS**, in analyse and study the data and find interesting relations. This should help us make predictions using the later **MAILOUT** files. Each row contains information about a single person, including their household, building and neighbourhood.

In the case of the two **MAILOUT** files, the **TRAIN** file contains a column called “RESPONSE”, which indicates whether that specific person is a good target for our mail order or not. The **TEST,** however, has that column removed and we would need to send to the following kaggle link in order to determine the final accuracy of the model:

https://www.kaggle.com/c/udacity-arvato-identify-customers/data

There are also two more files, outside the four that were originally mentioned, that help explain the columns available in the previous files:

DIAS Information Levels - Attributes 2017.xlsx: is an overview of the attributes and descriptions, sorted by the informational category.
DIAS Attributes - Values 2017.xlsx: provides a bit more detailed explanation of each feature's data values.

Problem Statement

Arvato wants first to analyze the attributes of existing clients and match them against a bigger dataset full of people in Germany. This way, we can sort out the customers that have similar behaviour and statistics into specific groups, making it easier to analyse and later perform predictions.

Secondly, we need to find out which people in Germany are most likely to be new customers willing to buy organic products through a specific mail order. So, instead of reaching out to the entire population of Germany, we can just target those specific people that would be interested.

To accomplish these tasks, we will spend a bit of time getting to know the data, finding any interesting patterns and visualizations that would help us gain some simple domain knowledge on the subject.

We also need to find ways to deal with any missing data, by either removing columns completely, or find ways to fill them in (mean, mode, etc). After we're done, we will scale the data to be numbers between 0 and 1.

We will use a specific dimensionality reduction technique, called principal component analysis (PCA), because the columns prove to be too many. The main goal of this step is to make sure the data is thoroughly cleaned and ready enough so as to not to confuse any machine learning algorithms we would use in the future.

Next, we will use the classic clustering analysis algorithm in order to group up related or similar customers into a specific number of groups. This number will be determined by visualizing different cluster numbers and finding the optimum number using the 'elbow' method.

After that, we will use the Train data already provided to train our machine learning algorithms. We might need to further split up the train into a train and validation set, to help later optimize our algorithms. The optimization would be done using a Bayesian Search to tune the many different hyperparameters, possibly using the full train data, as grid search does its own data split into validation.

Finally, after the model is ready and trained, we will use it to make predictions on the Test data provided and send the result to kaggle in order to determine how well our model performed.

Metrics

For PCA, we will find the optimum number of components that accomplishes 75% total explained variance.

For clustering, we will plot the different numbers of clusters onto a graph and find the ‘elbow’ point that tells us the appropriate number of clusters to use.

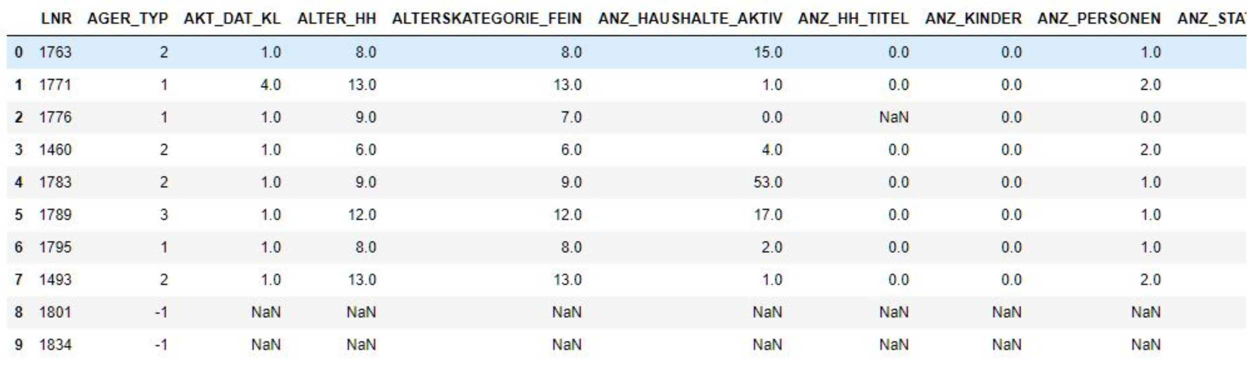
For machine learning classifiers, we will use AUC; **A**rea **U**nder the **R**eceiver operating characteristic curve. This is the metric that the Kaggle competition will be using to evaluate our model over the test dataset.

We will also take a look at the confusion matrix to understand if our model is predicting the 1s and 0s in a balanced matter.

II. Analysis

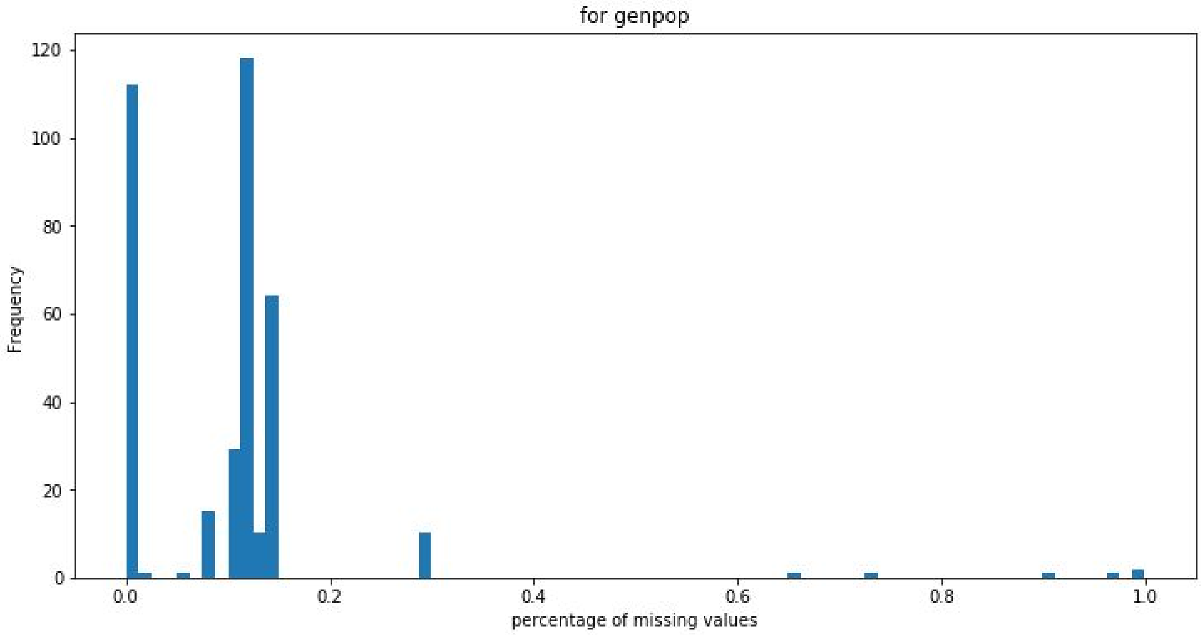
Data Exploration

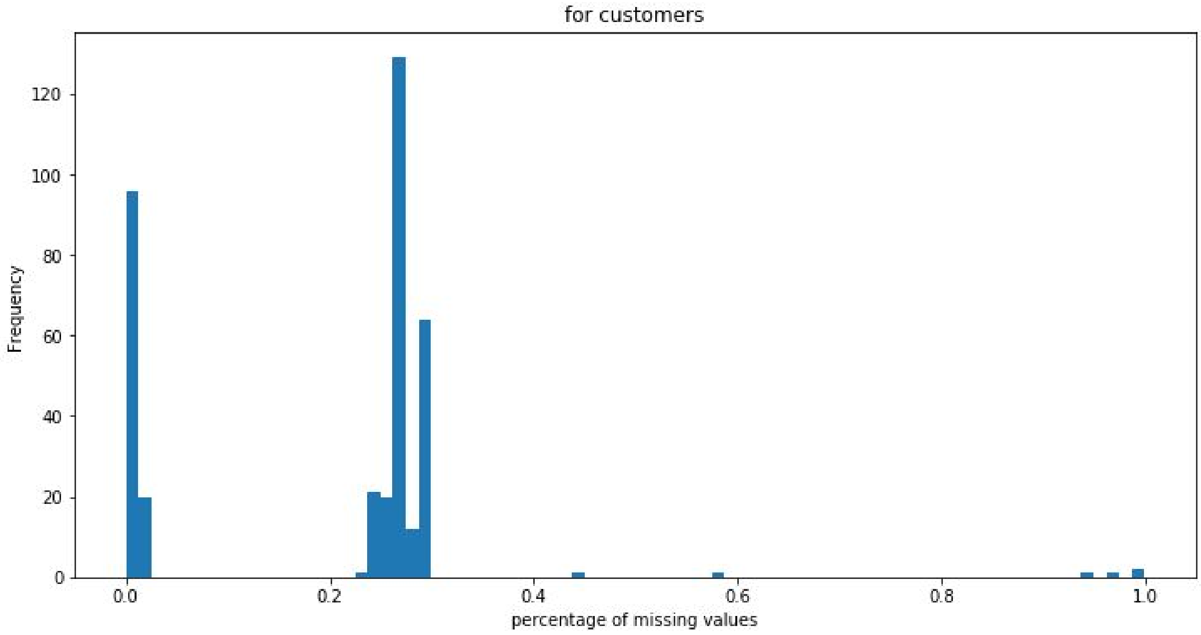
We’ll take a look at the data and they are all mostly following the same format and looks as shown below.

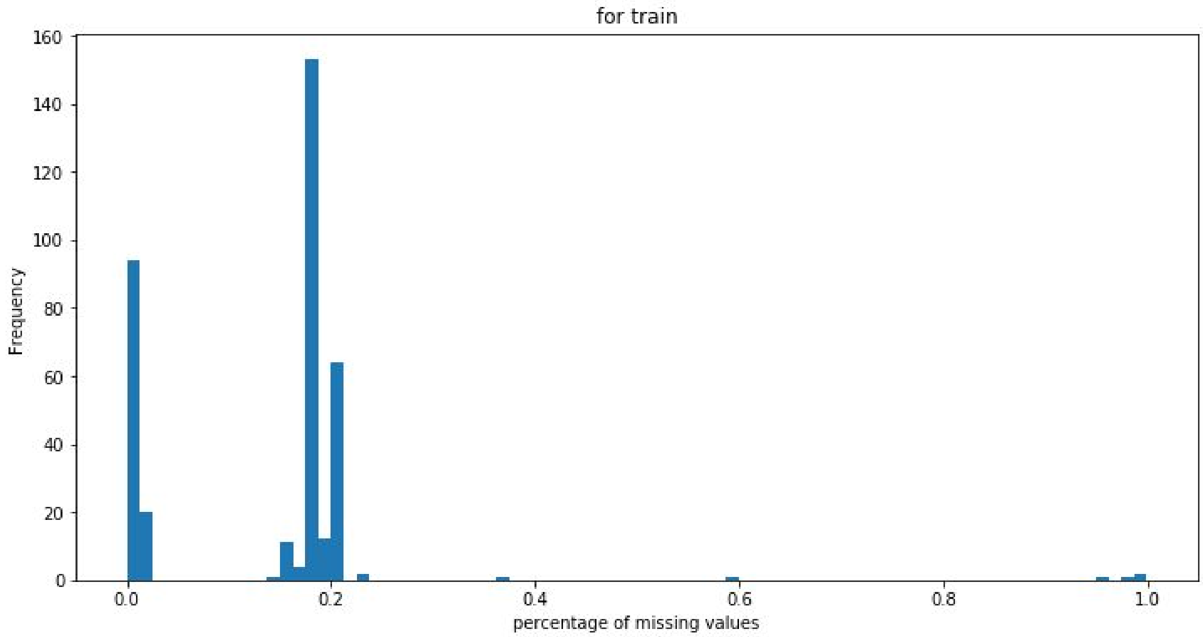


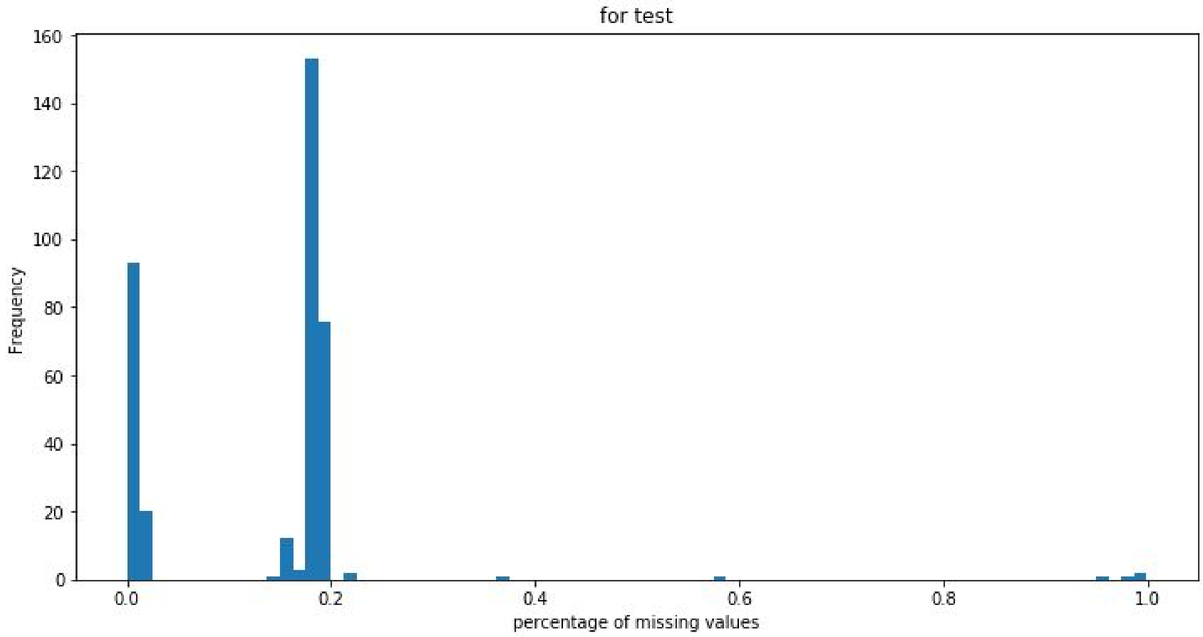
	LNR	AGER_TYP	AKT_DAT_KL	ALTER_HH	ALTERSKATEGORIE_FEIN	ANZ_HAUSHALTE_AKTIV	ANZ_HH_TITEL	ANZ_KINDER	ANZ_PERSONEN	ANZ_STA
0	1763	2	1.0	8.0	8.0	15.0	0.0	0.0	1.0	
1	1771	1	4.0	13.0	13.0	1.0	0.0	0.0	2.0	
2	1776	1	1.0	9.0	7.0	0.0	NaN	0.0	0.0	
3	1460	2	1.0	6.0	6.0	4.0	0.0	0.0	2.0	
4	1783	2	1.0	9.0	9.0	53.0	0.0	0.0	1.0	
5	1789	3	1.0	12.0	12.0	17.0	0.0	0.0	1.0	
6	1795	1	1.0	8.0	8.0	2.0	0.0	0.0	1.0	
7	1493	2	1.0	13.0	13.0	1.0	0.0	0.0	2.0	
8	1801	-1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
9	1834	-1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

We can already see a few NaNs there, so let’s take a look at how many there are over the datasets.

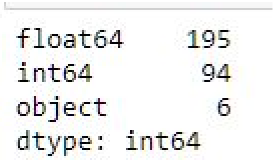

for genpop
Frequency
0
20
40
60
80
100
120
0.0
0.2
0.4
0.6
0.8
1.0
percentage of missing values


for customers
Frequency
0
20
40
60
80
100
120
0.0
0.2
0.4
0.6
0.8
1.0
percentage of missing values



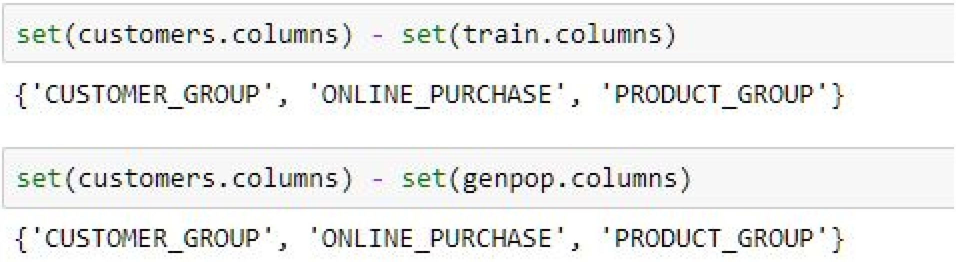


Looking at the datatypes:



```
float64    195
int64       94
object       6
dtype: int64
```

We will check customers dataset to see if it really has more columns than the others and what they are:



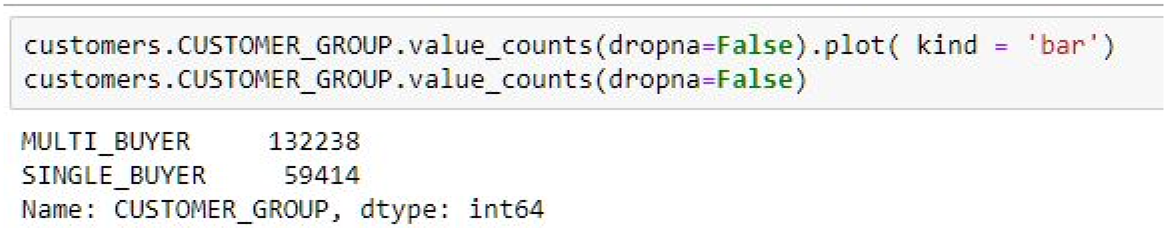
```
set(customers.columns) - set(train.columns)
```

```
{'CUSTOMER_GROUP', 'ONLINE_PURCHASE', 'PRODUCT_GROUP'}
```

```
set(customers.columns) - set(genpop.columns)
```

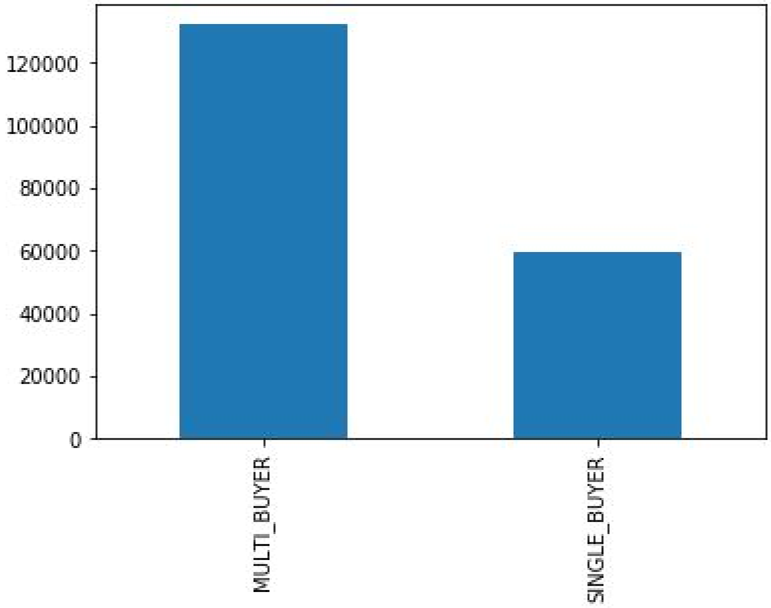
```
{'CUSTOMER_GROUP', 'ONLINE_PURCHASE', 'PRODUCT_GROUP'}
```

Let’s take a look at each of these to see how they are.
First, we’ll look at the CUSTOMER_GROUP:

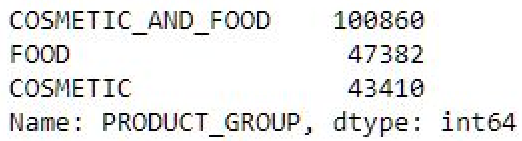


```
customers.CUSTOMER_GROUP.value_counts(dropna=False).plot( kind = 'bar')
customers.CUSTOMER_GROUP.value_counts(dropna=False)
```

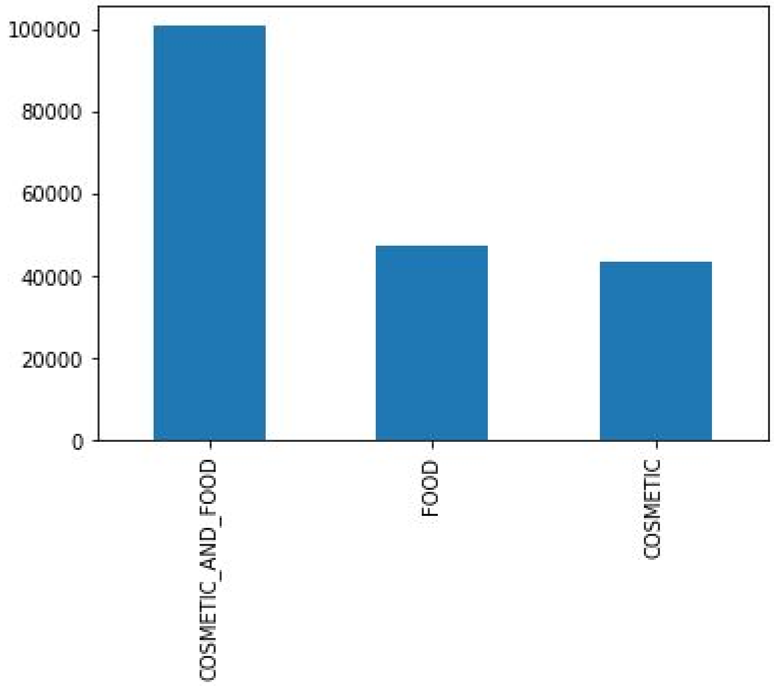
```
MULTI_BUYER     132238
SINGLE_BUYER     59414
Name: CUSTOMER_GROUP, dtype: int64
```



We can see that the customers dataset is about ⅔ Multi_Buyer and ⅓ Single_buyer.
Let’s take a look at the PRODUCT_GROUP:

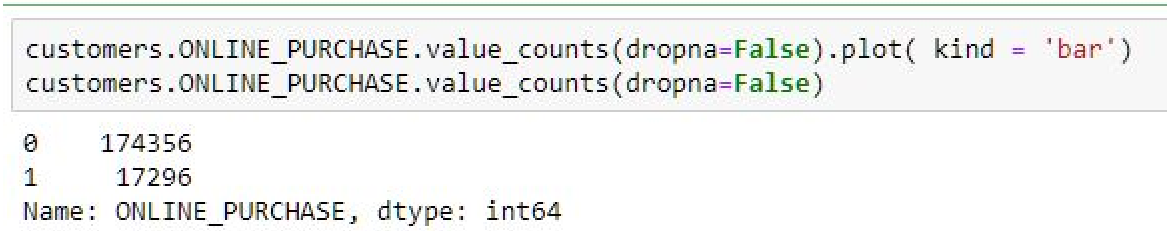


```
COSMETIC_AND_FOOD    100860
FOOD                  47382
COSMETIC              43410
Name: PRODUCT_GROUP, dtype: int64
```



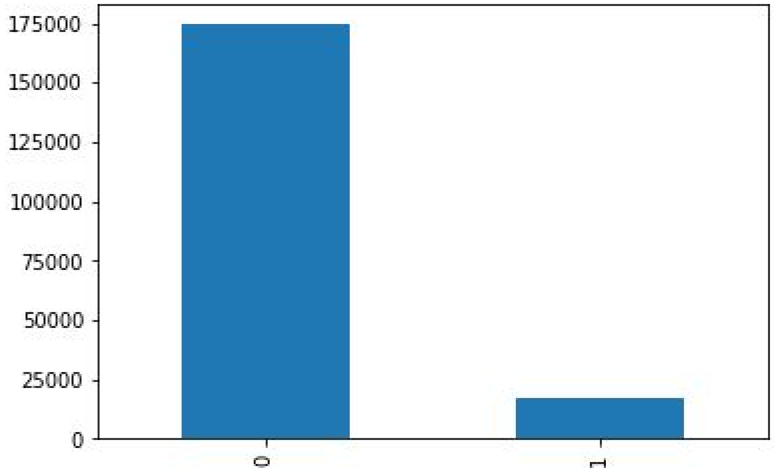
This one is made up of about ¼ food, ¼ cosmetic and the last half is of people purchasing both food and cosmetic.

Let’s look at ONLINE_PURCHASE:



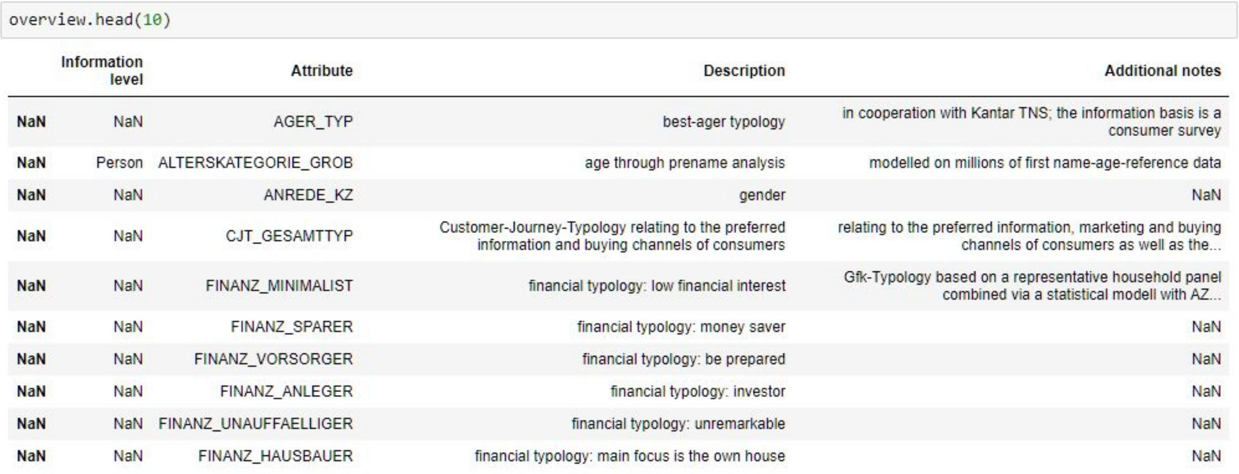
```
customers.ONLINE_PURCHASE.value_counts(dropna=False).plot( kind = 'bar')
customers.ONLINE_PURCHASE.value_counts(dropna=False)
```

```
0    174356
1     17296
Name: ONLINE_PURCHASE, dtype: int64
```



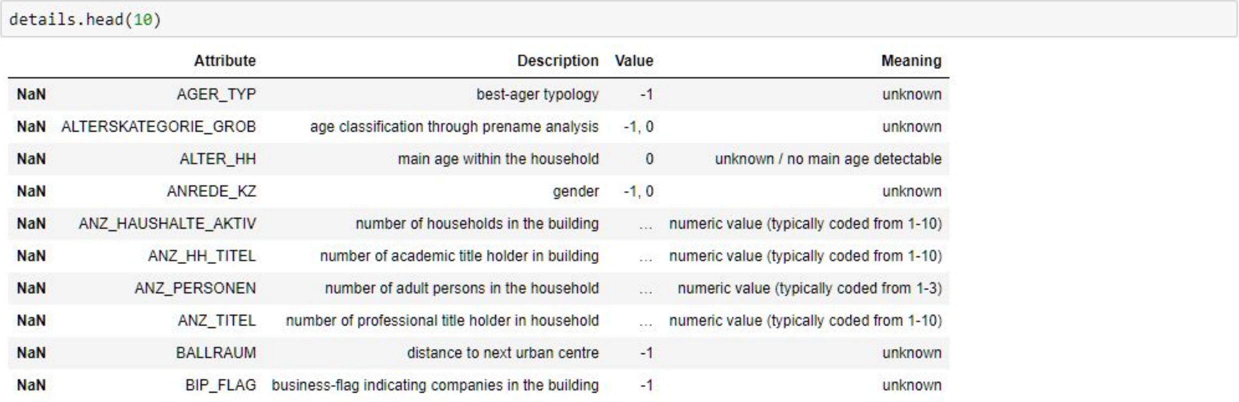
This one is made up almost completely of 0 for online purchase, suggesting that maybe these purchases were made offline.
While this seems very interesting, we can't use these features as they are not available in our other datasets.

We can try to look at our informational datasets to try to understand more about our features:



```
overview.head(10)
```

	Information level	Attribute	Description	Additional notes
NaN	NaN	AGER_TYP	best-ager typology	in cooperation with Kantar TNS; the information basis is a consumer survey
NaN	Person	ALTERSKATEGORIE_GROB	age through prename analysis	modelled on millions of first name-age-reference data
NaN	NaN	ANREDE_KZ	gender	NaN
NaN	NaN	CJT_GESAMTTYP	Customer-Journey-Typology relating to the preferred information and buying channels of consumers	relating to the preferred information, marketing and buying channels of consumers as well as the...
NaN	NaN	FINANZ_MINIMALIST	financial typology: low financial interest	Gfk-Typology based on a representative household panel combined via a statistical modell with AZ...
NaN	NaN	FINANZ_SPARER	financial typology: money saver	NaN
NaN	NaN	FINANZ_VORSORGER	financial typology: be prepared	NaN
NaN	NaN	FINANZ_ANLEGER	financial typology: investor	NaN
NaN	NaN	FINANZ_UNAUFFAELLIGER	financial typology: unremarkable	NaN
NaN	NaN	FINANZ_HAUSBAUER	financial typology: main focus is the own house	NaN



```
details.head(10)
```

	Attribute	Description	Value	Meaning
NaN	AGER_TYP	best-ager typology	-1	unknown
NaN	ALTERSKATEGORIE_GROB	age classification through prename analysis	-1, 0	unknown
NaN	ALTER_HH	main age within the household	0	unknown / no main age detectable
NaN	ANREDE_KZ	gender	-1, 0	unknown
NaN	ANZ_HAUSHALTE_AKTIV	number of households in the building	...	numeric value (typically coded from 1-10)
NaN	ANZ_HH_TITEL	number of academic title holder in building	...	numeric value (typically coded from 1-10)
NaN	ANZ_PERSONEN	number of adult persons in the household	...	numeric value (typically coded from 1-3)
NaN	ANZ_TITEL	number of professional title holder in household	...	numeric value (typically coded from 1-10)
NaN	BALLRAUM	distance to next urban centre	-1	unknown
NaN	BIP_FLAG	business-flag indicating companies in the building	-1	unknown

Some examples of some of the fields in more detail:

AGER_TYP: best-ager typology

ALTERSKATEGORIE_GROB: age through prename analysis

ALTER_HH: main age within the household

ANREDE_KZ: gender

ANZ_HAUSHALTE_AKTIV: number of households known in this building

D19_BANKEN_ANZ_12: transaction activity BANKS in the last 12 months

D19_BANKEN_ANZ_24: transaction activity BANKS in the last 24 months

D19_BANKEN_DATUM: actuality of the last transaction for the segment banks TOTAL

D19_VERSAND_DATUM: actuality of the last transaction for the segment mail-order TOTAL

FINANZ_ANLEGER: financial typology: investor

FINANZ_HAUSBAUER: financial typology: main focus is the own house

FINANZ_MINIMALIST: financial typology: low financial interest

KBA13_ALTERHALTER_30: share of car owners below 31 within the PLZ8

KBA13_ALTERHALTER_45: share of car owners between 31 and 45 within the PLZ8

KBA13_ALTERHALTER_60: share of car owners between 46 and 60 within the PLZ8

KBA13_ALTERHALTER_61: share of car owners elder than 60 within the PLZ8
KLP_FAMILIE_GROB: family type rough

LP_LEBENSPHASE_FEIN: lifestage fine

LP_LEBENSPHASE_GROB: lifestage rough

LP_STATUS_FEIN: social status fine

PLZ8_ANTG2: number of 3-5 family houses in the PLZ8

PLZ8_ANTG3: number of 6-10 family houses in the PLZ8

PLZ8_ANTG4: number of >10 family houses in the PLZ8

SEMIO_KAEM: affinity indicating in what way the person is of a fightfull attitude

SEMIO_KRIT: affinity indicating in what way the person is critical minded

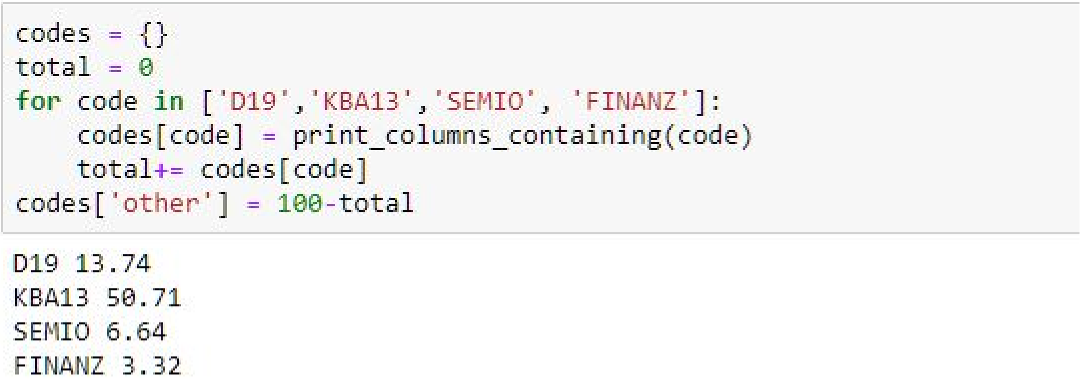
SEMIO_KULT: affinity indicating in what way the person is cultural minded

SEMIO_LUST: affinity indicating in what way the person is sensual minded

SEMIO_MAT: affinity indicating in what way the person is material minded

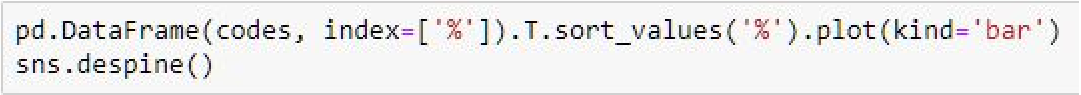
After manually reading through them, we can see that most of the columns consist of the following information:

- household
- transactional activity (columns starting with D19)
- Cars (columns starting with KBA13)
- PLZ8 demographics
- financial typology(columns starting with FINANZ)
- mindedness (columns starting with SEMIO)

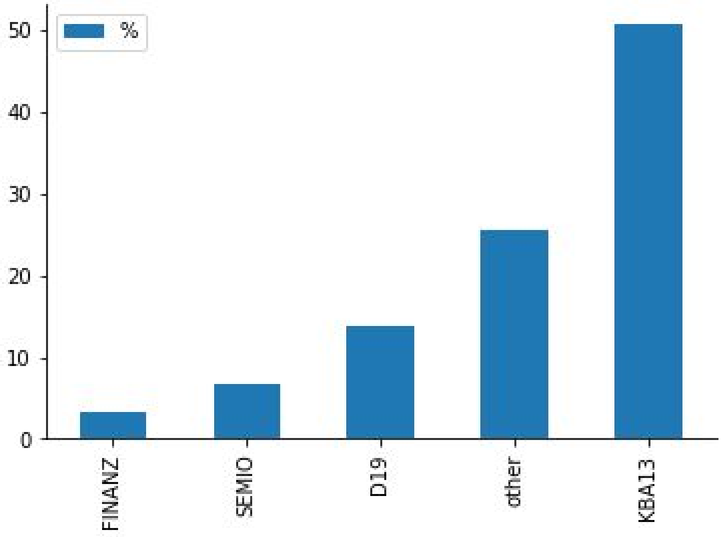


```
codes = {}
total = 0
for code in ['D19','KBA13','SEMIO', 'FINANZ']:
    codes[code] = print_columns_containing(code)
    total+= codes[code]
codes['other'] = 100-total
```

```
D19 13.74
KBA13 50.71
SEMIO 6.64
FINANZ 3.32
```



```
pd.DataFrame(codes, index=['%']).T.sort_values('%').plot(kind='bar')
sns.despine()
```

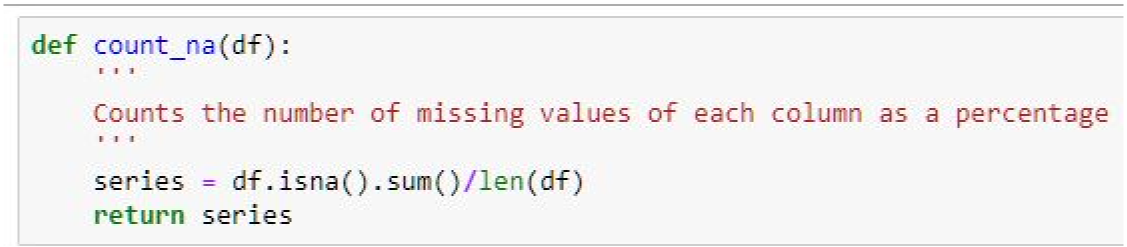


We can see that a huge number of the columns (over 50% of the train dataset) is car data.

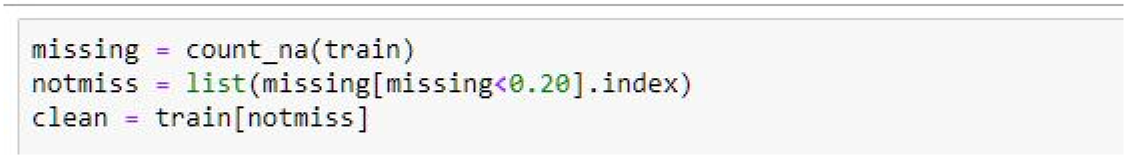
Transactional data, which is probably more relevant than cars to our project, consists of only 13.74% of the columns available in the train dataset.

III. Data Preprocessing

First step is dealing with the empty values.

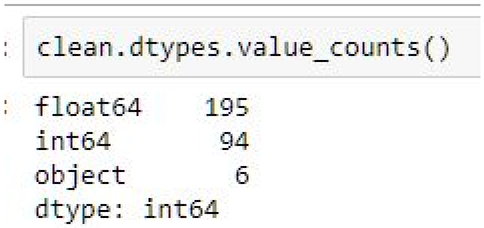


```
def count_na(df):
    '''
    Counts the number of missing values of each column as a percentage
    '''
    series = df.isna().sum()/len(df)
    return series
```



```
missing = count_na(train)
notmiss = list(missing[missing<0.20].index)
clean = train[notmiss]
```

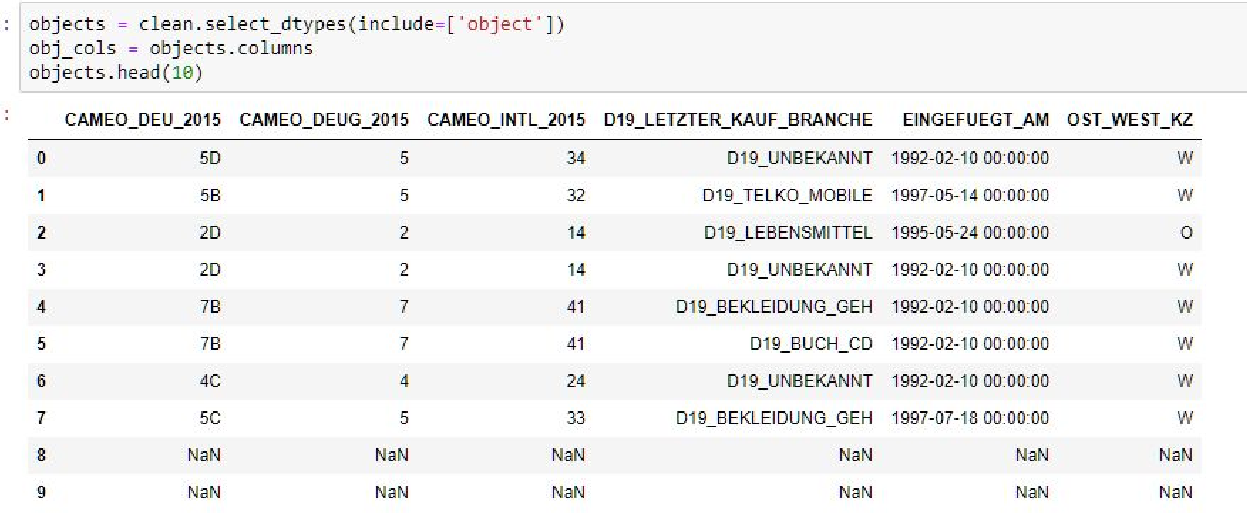
We choose to remove all rows that have more than 20% missing columns.
Then, we take a look at what's left:



```
: clean.dtypes.value_counts()

: float64    195
  int64       94
  object       6
  dtype: int64
```

Cleaning up Categorical Features

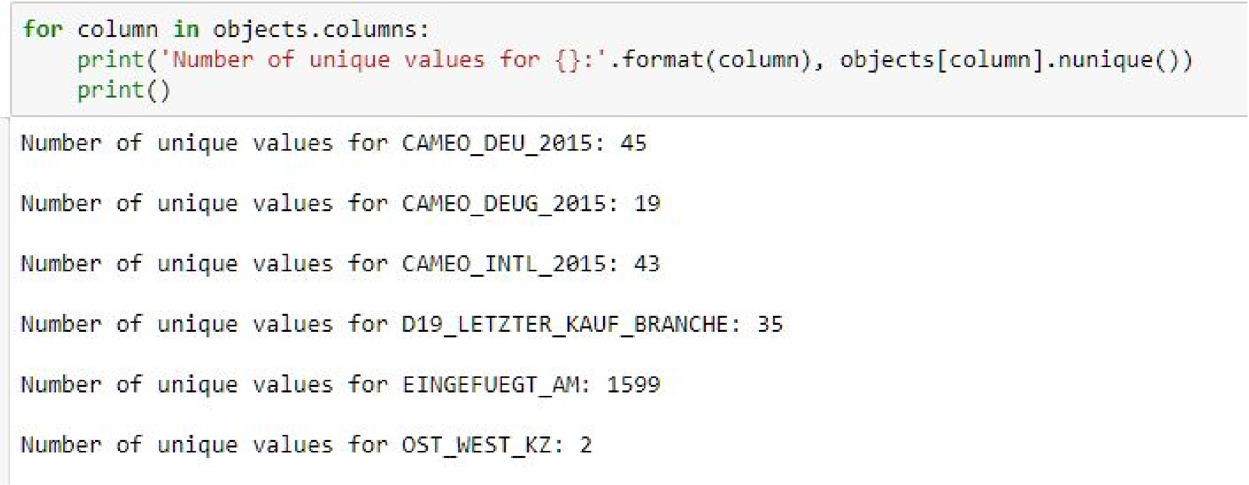
We will deal with objects first.



```
: objects = clean.select_dtypes(include=['object'])
  obj_cols = objects.columns
  objects.head(10)
```

	CAMEO_DEU_2015	CAMEO_DEUG_2015	CAMEO_INTL_2015	D19_LETZTER_KAUF_BRANCHE	EINGEFUEGT_AM	OST_WEST_KZ
0	5D	5	34	D19_UNBEKANNT	1992-02-10 00:00:00	W
1	5B	5	32	D19_TELKO_MOBILE	1997-05-14 00:00:00	W
2	2D	2	14	D19_LEBENSMITTEL	1995-05-24 00:00:00	O
3	2D	2	14	D19_UNBEKANNT	1992-02-10 00:00:00	W
4	7B	7	41	D19_BEKLEIDUNG_GEH	1992-02-10 00:00:00	W
5	7B	7	41	D19_BUCH_CD	1992-02-10 00:00:00	W
6	4C	4	24	D19_UNBEKANNT	1992-02-10 00:00:00	W
7	5C	5	33	D19_BEKLEIDUNG_GEH	1997-07-18 00:00:00	W
8	NaN	NaN	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN	NaN	NaN

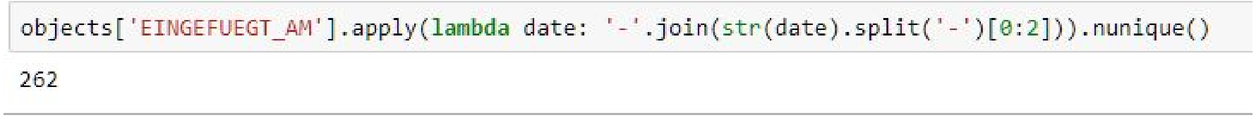
For objects, we generally want to fill in the nans by the most frequent value. After that, we will want to convert them all to numbers. We do that by creating dummy variables for each unique value per feature. So, let's look at the number of unique values first:



```
for column in objects.columns:
    print('Number of unique values for {}:'.format(column), objects[column].nunique())
    print()
```

```
Number of unique values for CAMEO_DEU_2015: 45

Number of unique values for CAMEO_DEUG_2015: 19

Number of unique values for CAMEO_INTL_2015: 43

Number of unique values for D19_LETZTER_KAUF_BRANCHE: 35

Number of unique values for EINGEFUEGT_AM: 1599

Number of unique values for OST_WEST_KZ: 2
```

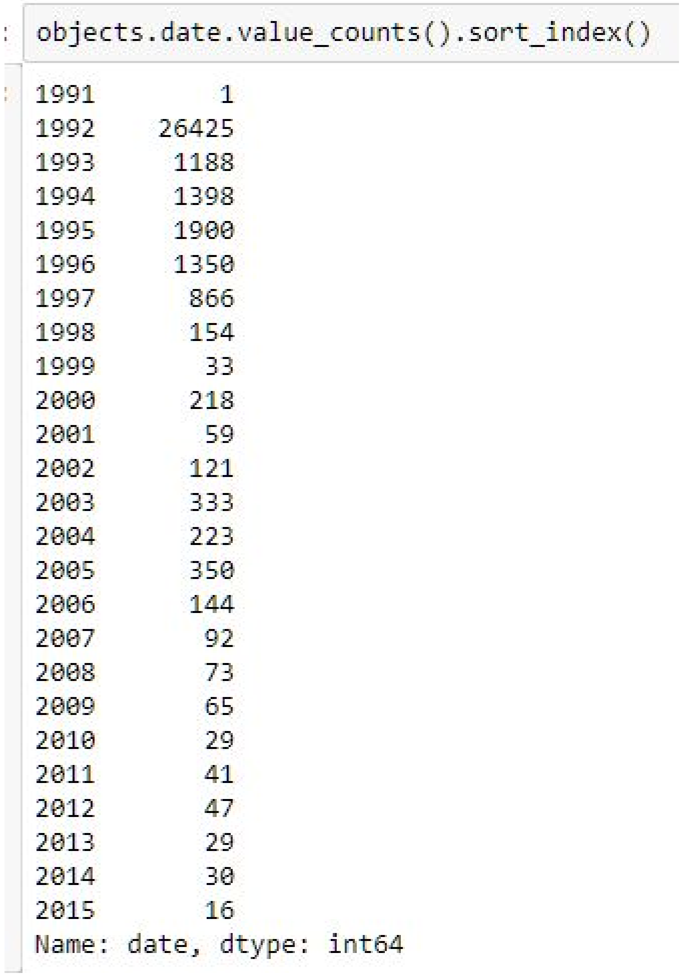
Since EINGEFUEGT_AM has so many unique values and after looking at the format, it clearly seems to be a date feature. To deal with this, we will try to spit it up into a Year-Month format.



```
objects['EINGEFUEGT_AM'].apply(lambda date: '-'.join(str(date).split('-')[0:2])).nunique()
```

```
262
```

Looking at the uniques after that, we see that there are still 262 values. These are too many to use as dummy variables. So, we will have to change them to just a Year format. We’ll also give it a suitable name: date.

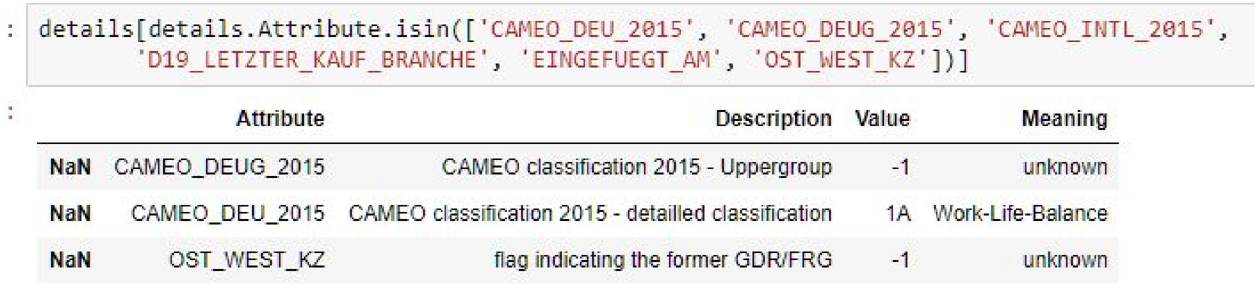


```python
objects.date.value_counts().sort_index()
```

```
1991        1
1992    26425
1993     1188
1994     1398
1995     1900
1996     1350
1997      866
1998      154
1999       33
2000      218
2001       59
2002      121
2003      333
2004      223
2005      350
2006      144
2007       92
2008       73
2009       65
2010       29
2011       41
2012       47
2013       29
2014       30
2015       16
Name: date, dtype: int64
```

Looking at the counts of each unique year, we see that the data is split up over 25 years, with only 1 value in the first year (1991), and last year being 2015.

Moving on, let's try to understand what some of the other features mean.

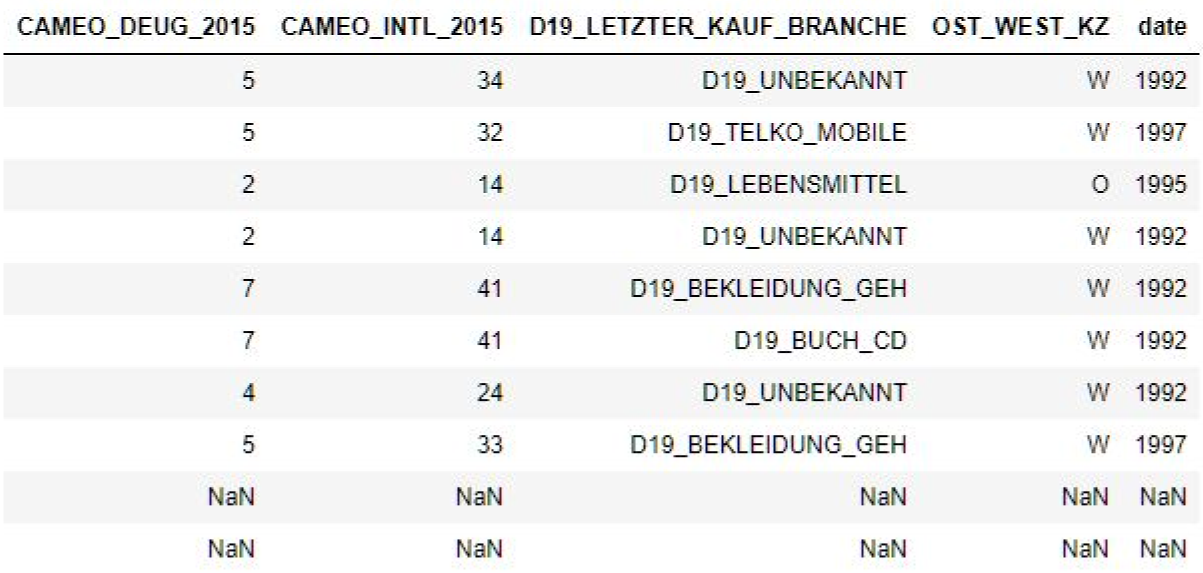


```python
details[details.Attribute.isin(['CAMEO_DEU_2015', 'CAMEO_DEUG_2015', 'CAMEO_INTL_2015',
        'D19_LETZTER_KAUF_BRANCHE', 'EINGEFUEGT_AM', 'OST_WEST_KZ'])]
```

	Attribute	Description	Value	Meaning
NaN	CAMEO_DEUG_2015	CAMEO classification 2015 - Uppergroup	-1	unknown
NaN	CAMEO_DEU_2015	CAMEO classification 2015 - detailled classification	1A	Work-Life-Balance
NaN	OST_WEST_KZ	flag indicating the former GDR/FRG	-1	unknown

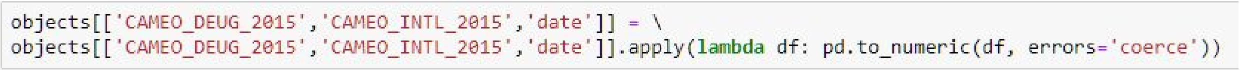
We could not get details of all the columns, but we can see that CAMEO_DEU is the same as CAMEO_DEUG, so we can just remove one of them. We'll choose to keep CAMEO_DEUG, since it has less unique values.

Now, for object type of data, we generally want to fill in the nans by the most frequent value. Let’s see how our dataframe looks now:



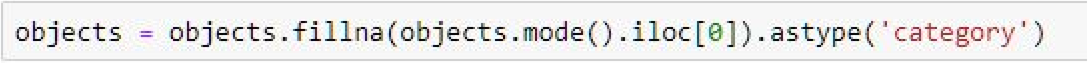
CAMEO_DEUG_2015	CAMEO_INTL_2015	D19_LETZTER_KAUF_BRANCHE	OST_WEST_KZ	date
5	34	D19_UNBEKANNT	W	1992
5	32	D19_TELKO_MOBILE	W	1997
2	14	D19_LEBENSMITTEL	O	1995
2	14	D19_UNBEKANNT	W	1992
7	41	D19_BEKLEIDUNG_GEH	W	1992
7	41	D19_BUCH_CD	W	1992
4	24	D19_UNBEKANNT	W	1992
5	33	D19_BEKLEIDUNG_GEH	W	1997
NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN

We want to make sure that all our numbers-like columns actually contain only numbers and no letters or words in them. So, we will convert the non-numbers to NaNs using the following:



```
objects[['CAMEO_DEUG_2015','CAMEO_INTL_2015','date']] = \
objects[['CAMEO_DEUG_2015','CAMEO_INTL_2015','date']].apply(lambda df: pd.to_numeric(df, errors='coerce'))
```

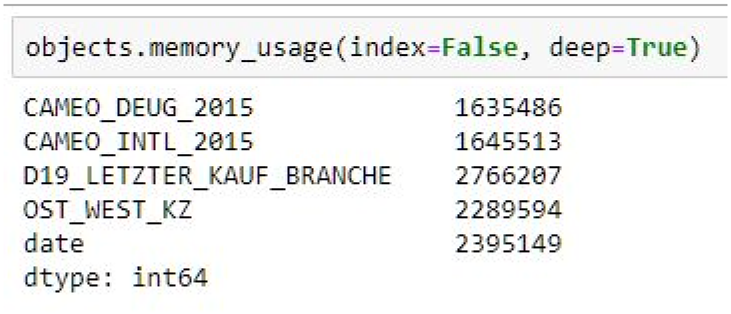
Now we are ready to change all our missing fields to mode, the most frequent value. We do this using the following step:



```
objects = objects.fillna(objects.mode().iloc[0]).astype('category')
```

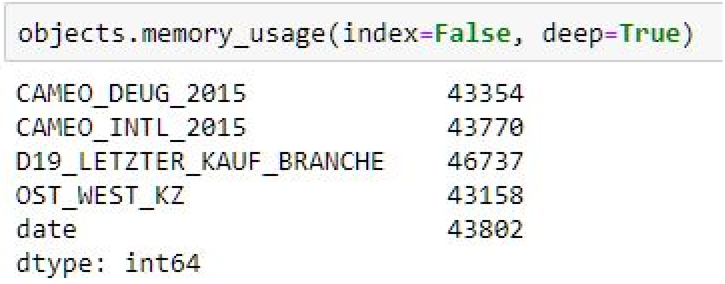
Note that because our data is so large, it seems to be taking a heavy load on the memory. So, we changed the data type of these columns to be category after filling in the missing values.

Memory used would decrease from:



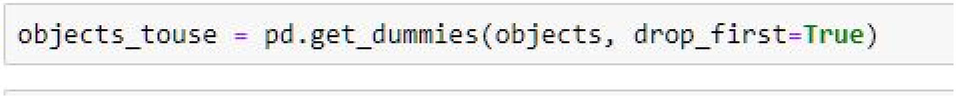
```
objects.memory_usage(index=False, deep=True)

CAMEO_DEUG_2015             1635486
CAMEO_INTL_2015             1645513
D19_LETZTER_KAUF_BRANCHE    2766207
OST_WEST_KZ                 2289594
date                        2395149
dtype: int64
```

to:

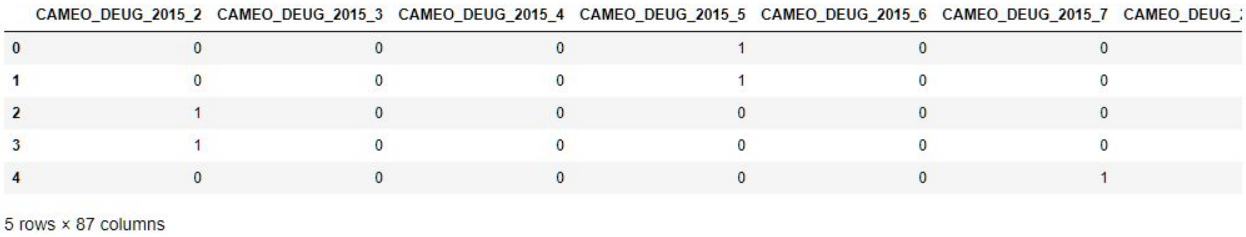


```
objects.memory_usage(index=False, deep=True)

CAMEO_DEUG_2015             43354
CAMEO_INTL_2015             43770
D19_LETZTER_KAUF_BRANCHE    46737
OST_WEST_KZ                 43158
date                        43802
dtype: int64
```

Finally, we will want to convert them all to numbers.
We do that by creating dummy variables for each unique value per feature through the following cell:



```
objects_touse = pd.get_dummies(objects, drop_first=True)
```

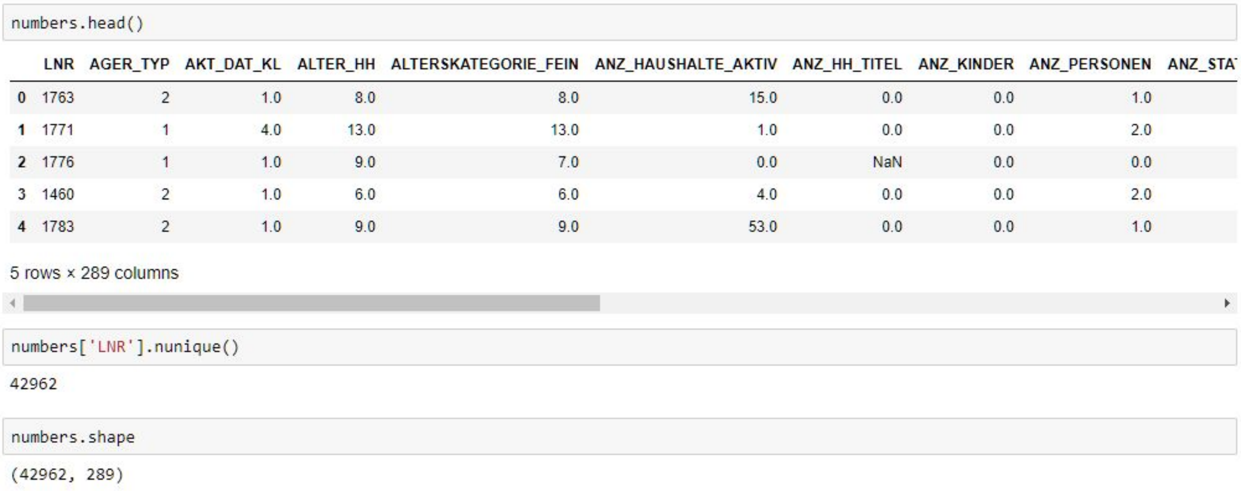


	CAMEO_DEUG_2015_2	CAMEO_DEUG_2015_3	CAMEO_DEUG_2015_4	CAMEO_DEUG_2015_5	CAMEO_DEUG_2015_6	CAMEO_DEUG_2015_7	CAMEO_DEUG_
0	0	0	0	1	0	0	
1	0	0	0	1	0	0	
2	1	0	0	0	0	0	
3	1	0	0	0	0	0	
4	0	0	0	0	0	1	

5 rows × 87 columns

Now our objects, or rather now categorical type, are ready to be used.

Cleaning up Numerical Features



```
numbers.head()
```

	LNR	AGER_TYP	AKT_DAT_KL	ALTER_HH	ALTERSKATEGORIE_FEIN	ANZ_HAUSHALTE_AKTIV	ANZ_HH_TITEL	ANZ_KINDER	ANZ_PERSONEN	ANZ_STA
0	1763	2	1.0	8.0	8.0	15.0	0.0	0.0	1.0	
1	1771	1	4.0	13.0	13.0	1.0	0.0	0.0	2.0	
2	1776	1	1.0	9.0	7.0	0.0	NaN	0.0	0.0	
3	1460	2	1.0	6.0	6.0	4.0	0.0	0.0	2.0	
4	1783	2	1.0	9.0	9.0	53.0	0.0	0.0	1.0	

5 rows × 289 columns

```
numbers['LNR'].nunique()
```

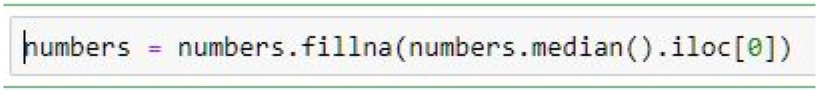
```
42962
```

```
numbers.shape
```

```
(42962, 289)
```

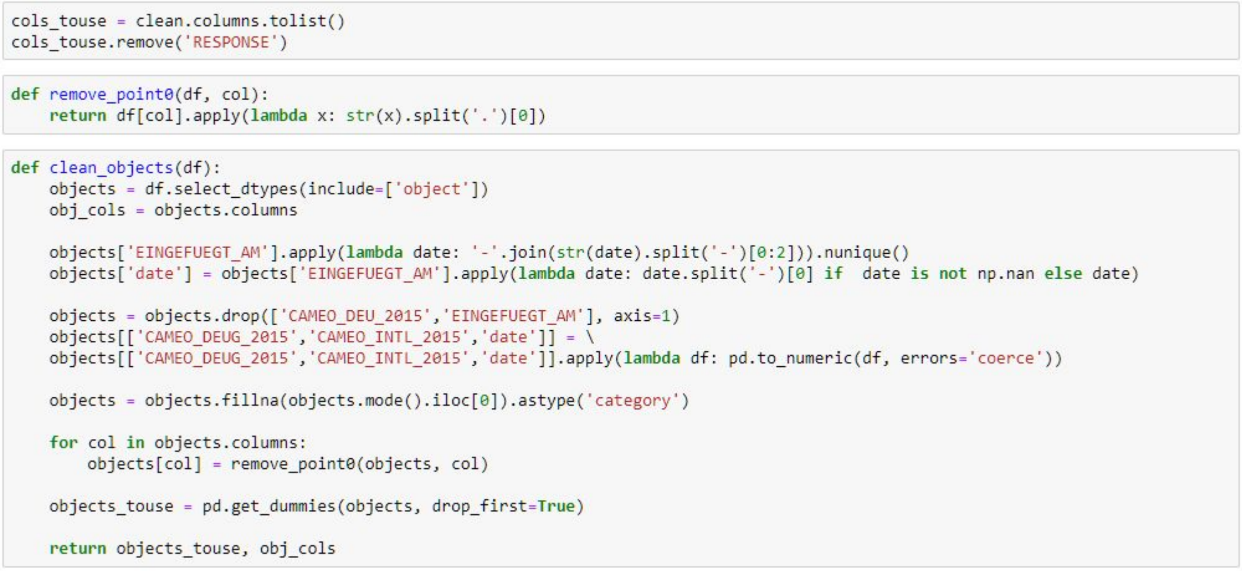
LNR is unique to each row so it is clearly an ID number. Since ids are meaningless to us, we'll remove it.

Numbers are much simpler to deal with than the categorical features. All we need to do is fill them by either mean or median. To avoid any effect by outliers, we chose median.



```
numbers = numbers.fillna(numbers.median().iloc[0])
```

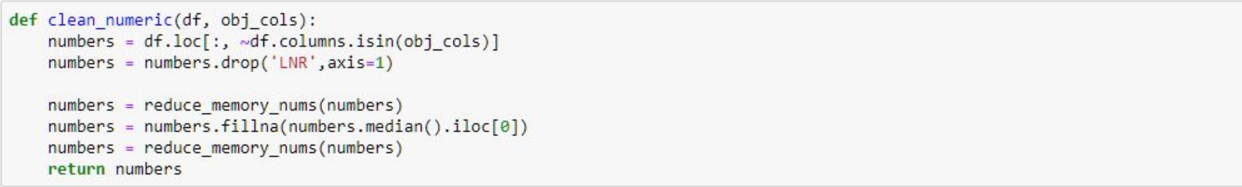
Now that we're done, let's wrap it all up into neat functions to be used for each dataset:



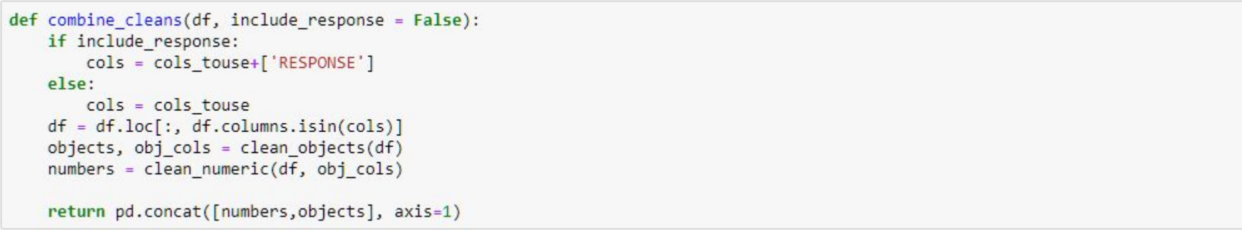
```
cols_touse = clean.columns.tolist()
cols_touse.remove('RESPONSE')
```

```
def remove_point0(df, col):
    return df[col].apply(lambda x: str(x).split('.')[0])
```

```
def clean_objects(df):
    objects = df.select_dtypes(include=['object'])
    obj_cols = objects.columns

    objects['EINGEFUEGT_AM'].apply(lambda date: '-'.join(str(date).split('-')[0:2])).nunique()
    objects['date'] = objects['EINGEFUEGT_AM'].apply(lambda date: date.split('-')[0] if  date is not np.nan else date)

    objects = objects.drop(['CAMEO_DEU_2015','EINGEFUEGT_AM'], axis=1)
    objects[['CAMEO_DEUG_2015','CAMEO_INTL_2015','date']] = \
    objects[['CAMEO_DEUG_2015','CAMEO_INTL_2015','date']].apply(lambda df: pd.to_numeric(df, errors='coerce'))

    objects = objects.fillna(objects.mode().iloc[0]).astype('category')

    for col in objects.columns:
        objects[col] = remove_point0(objects, col)

    objects_touse = pd.get_dummies(objects, drop_first=True)

    return objects_touse, obj_cols
```

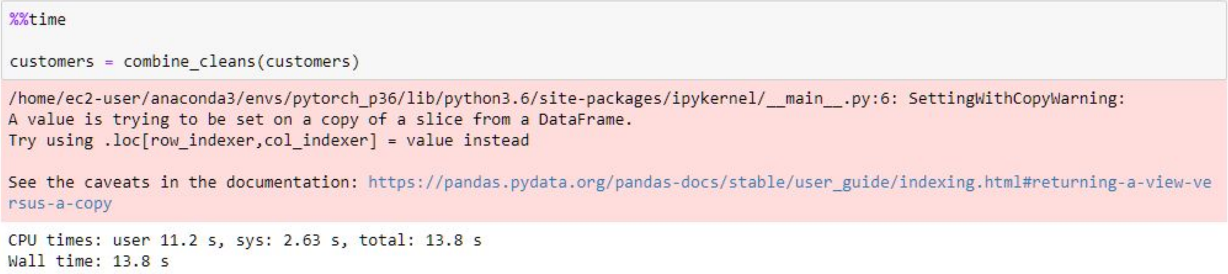


```python
def clean_numeric(df, obj_cols):
    numbers = df.loc[:, ~df.columns.isin(obj_cols)]
    numbers = numbers.drop('LNR',axis=1)

    numbers = reduce_memory_nums(numbers)
    numbers = numbers.fillna(numbers.median().iloc[0])
    numbers = reduce_memory_nums(numbers)
    return numbers
```



```python
def combine_cleans(df, include_response = False):
    if include_response:
        cols = cols_touse+['RESPONSE']
    else:
        cols = cols_touse
    df = df.loc[:, df.columns.isin(cols)]
    objects, obj_cols = clean_objects(df)
    numbers = clean_numeric(df, obj_cols)

    return pd.concat([numbers,objects], axis=1)
```

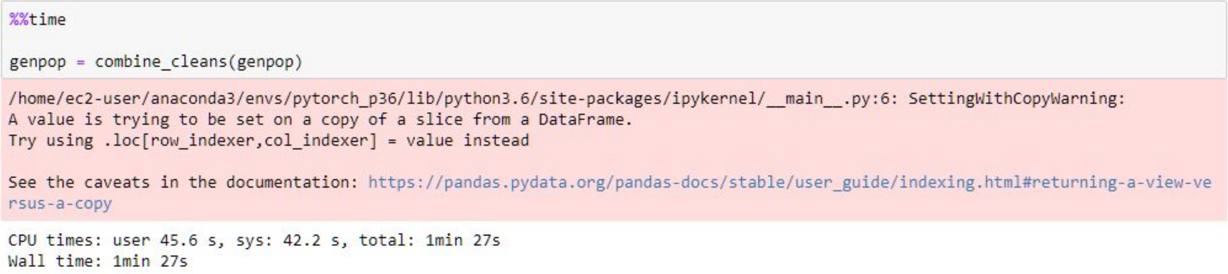
Then, we’ll apply these functions to each dataset (ignore the warnings):



```python
%%time

customers = combine_cleans(customers)
```

```
/home/ec2-user/anaconda3/envs/pytorch_p36/lib/python3.6/site-packages/ipykernel/__main__.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
```

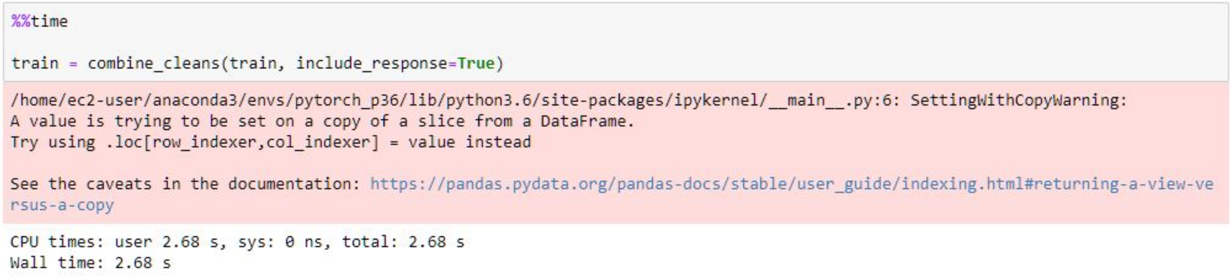
```
CPU times: user 11.2 s, sys: 2.63 s, total: 13.8 s
Wall time: 13.8 s
```



```python
%%time

genpop = combine_cleans(genpop)
```

```
/home/ec2-user/anaconda3/envs/pytorch_p36/lib/python3.6/site-packages/ipykernel/__main__.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
```

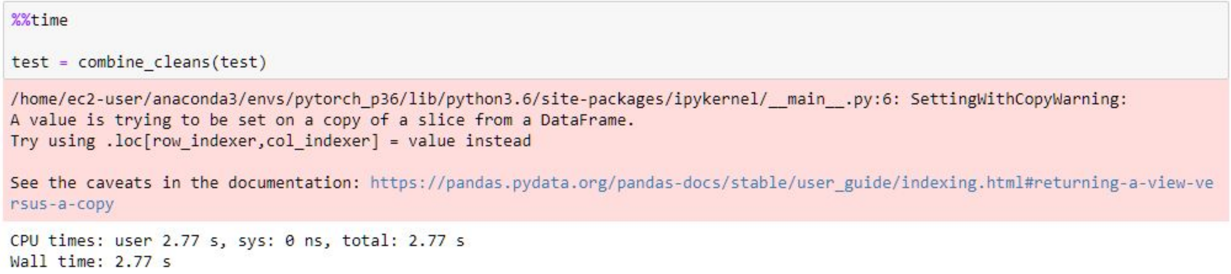
```
CPU times: user 45.6 s, sys: 42.2 s, total: 1min 27s
Wall time: 1min 27s
```



```
%%time

train = combine_cleans(train, include_response=True)
```

```
/home/ec2-user/anaconda3/envs/pytorch_p36/lib/python3.6/site-packages/ipykernel/__main__.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
```

```
CPU times: user 2.68 s, sys: 0 ns, total: 2.68 s
Wall time: 2.68 s
```



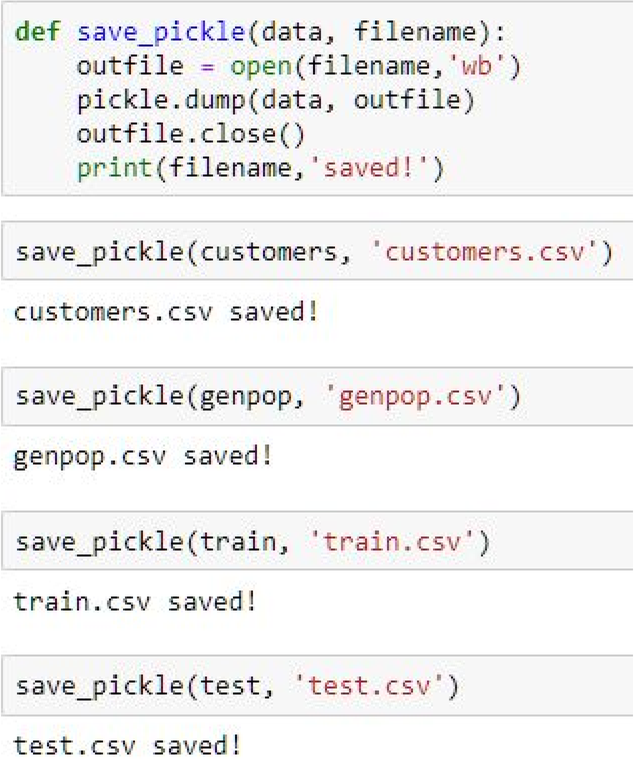
```
%%time

test = combine_cleans(test)
```

```
/home/ec2-user/anaconda3/envs/pytorch_p36/lib/python3.6/site-packages/ipykernel/__main__.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
```

```
CPU times: user 2.77 s, sys: 0 ns, total: 2.77 s
Wall time: 2.77 s
```

Finally, we’ll save our preprocessed data all into pickle files so that we don’t have to do all this preprocessing every time we come back to work on the project.

Note that we could have used joblib instead, however, joblib performs best with pure numpy arrays. We still want to keep the column names so we’ll keep them in pandas dataframe.



```
def save_pickle(data, filename):
    outfile = open(filename,'wb')
    pickle.dump(data, outfile)
    outfile.close()
    print(filename,'saved!')
```

```
save_pickle(customers, 'customers.csv')
```

```
customers.csv saved!
```

```
save_pickle(genpop, 'genpop.csv')
```

```
genpop.csv saved!
```

```
save_pickle(train, 'train.csv')
```

```
train.csv saved!
```

```
save_pickle(test, 'test.csv')
```

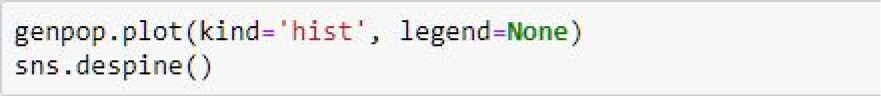
```
test.csv saved!
```

IV. Unsupervised Learning: Clustering

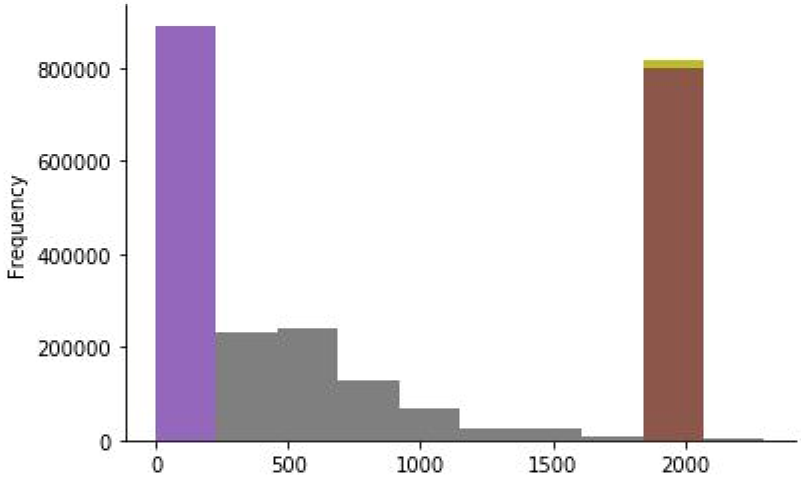
Preparation

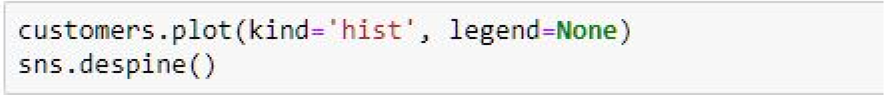
First thing we need to do here is to scale the data. We need to do this so that a feature from 1 to 10, for example, would be treated similar to one from 100 to 1000.

For this, we have 2 scalers to choose from: MinMaxScaler or Standard Scaler. Standard scaler is better suited if our data is normally distributed, otherwise MinMaxScaler is better.

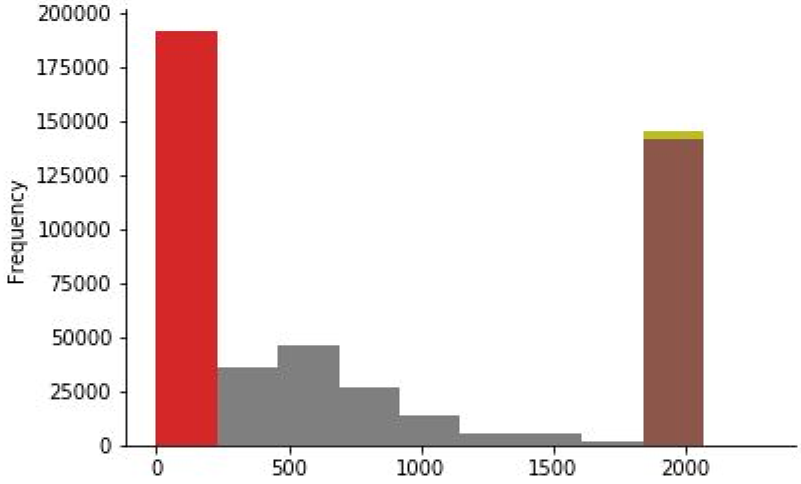


```
genpop.plot(kind='hist', legend=None)
sns.despine()
```

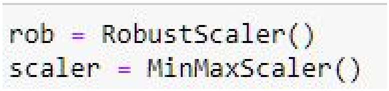




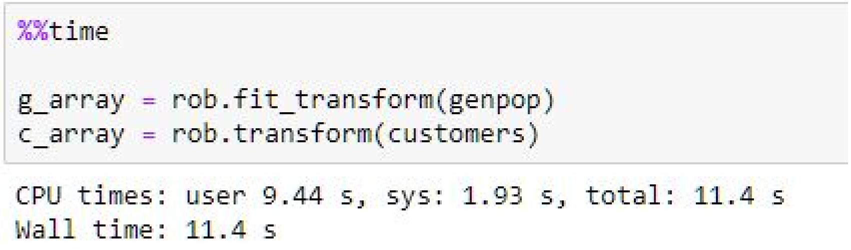
```
customers.plot(kind='hist', legend=None)
sns.despine()
```



A quick look at the distribution shows us that the data is not normally distributed at all. We also can see a lot of outliers. We will use Robust Scaler first though to get rid of these outliers.

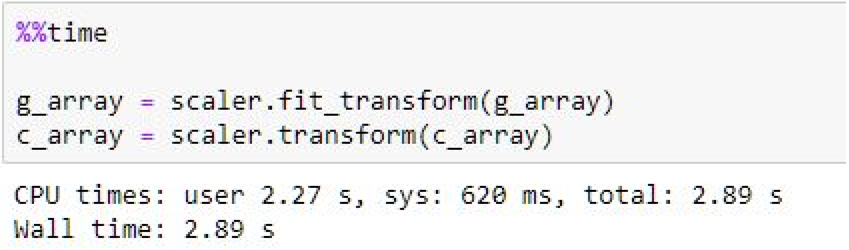


```
rob = RobustScaler()
scaler = MinMaxScaler()
```



```
%%time

g_array = rob.fit_transform(genpop)
c_array = rob.transform(customers)
```

```
CPU times: user 9.44 s, sys: 1.93 s, total: 11.4 s
Wall time: 11.4 s
```



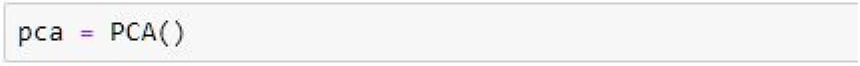
```
%%time

g_array = scaler.fit_transform(g_array)
c_array = scaler.transform(c_array)
```

```
CPU times: user 2.27 s, sys: 620 ms, total: 2.89 s
Wall time: 2.89 s
```

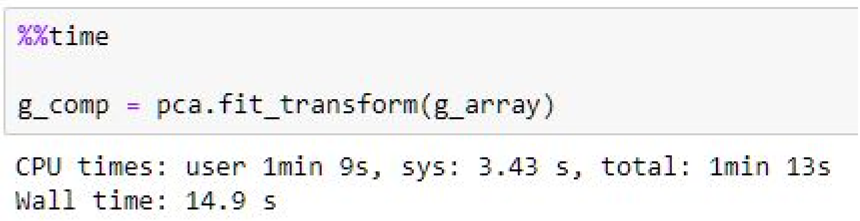
PCA

Because we have so many columns, we will need to reduce the dimensionality using PCA. Later on, our K-Means algorithm will also perform better this way.

Note that I have tried using Sagemaker’s PCA first and found it impossible to use. According to AWS documentation, there is a 60 second timeout when trying to use a model to predict. The problem is that our data is so large, that it needs more time than that for predictions. There is no possible way I could find to increase this limit. I even tried using the "ml.m5.4xlarge" which consists of a 16 core processor and 64 GiB Memory. Because of this, we will have to use SKLearn's PCA instead of Sagemaker's.



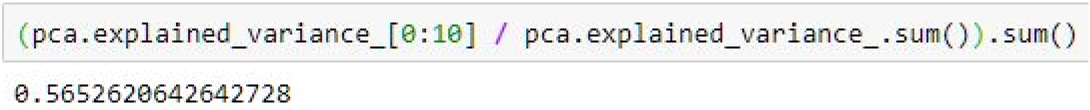
```
pca = PCA()
```



```
%%time

g_comp = pca.fit_transform(g_array)
```

```
CPU times: user 1min 9s, sys: 3.43 s, total: 1min 13s
Wall time: 14.9 s
```

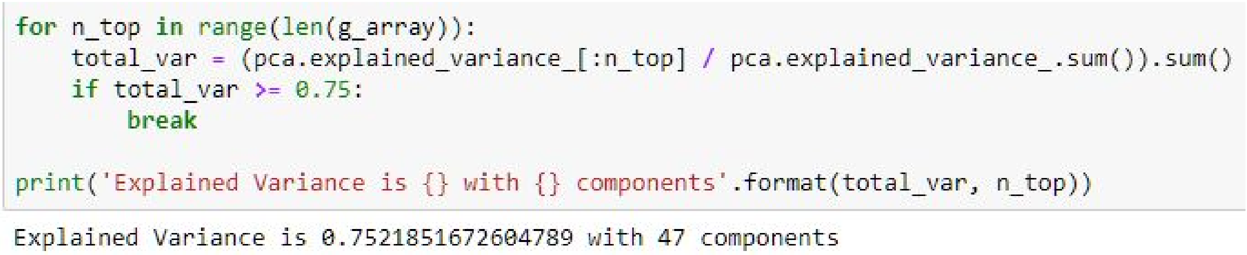
We are going by the strategy of using all components at first then reducing as many as we can till we get at about 75%.



```
(pca.explained_variance_[0:10] / pca.explained_variance_.sum()).sum()
```

```
0.5652620642642728
```

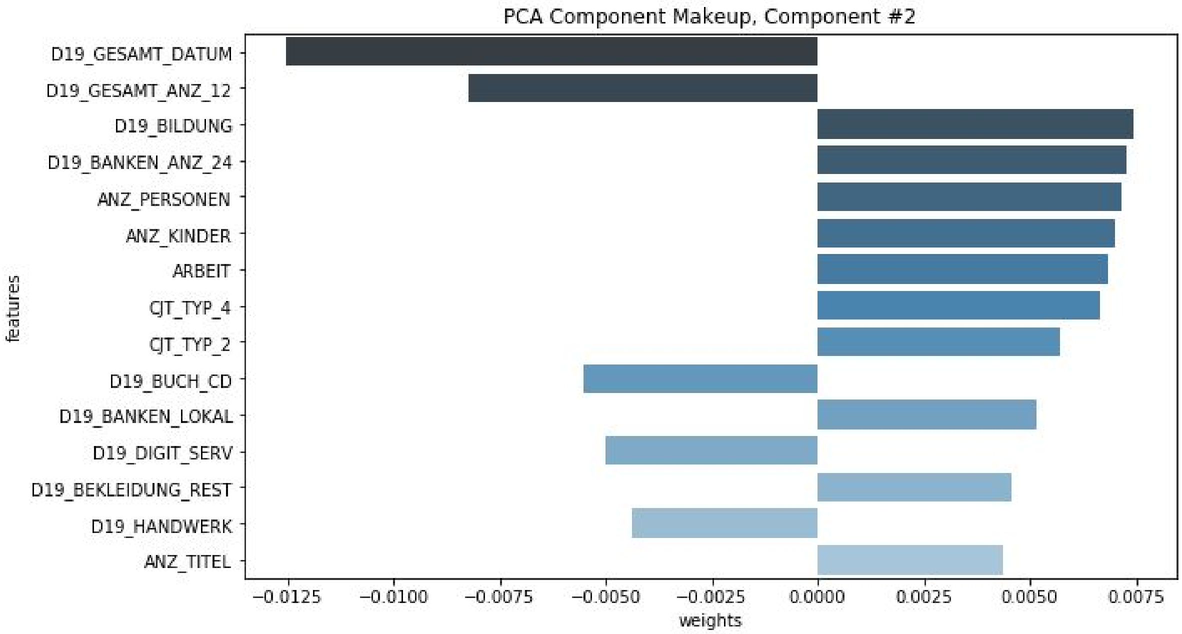
We can see that the top 10 components make up 56.5% of our total variance.



```
for n_top in range(len(g_array)):
    total_var = (pca.explained_variance_[:n_top] / pca.explained_variance_.sum()).sum()
    if total_var >= 0.75:
        break

print('Explained Variance is {} with {} components'.format(total_var, n_top))
```

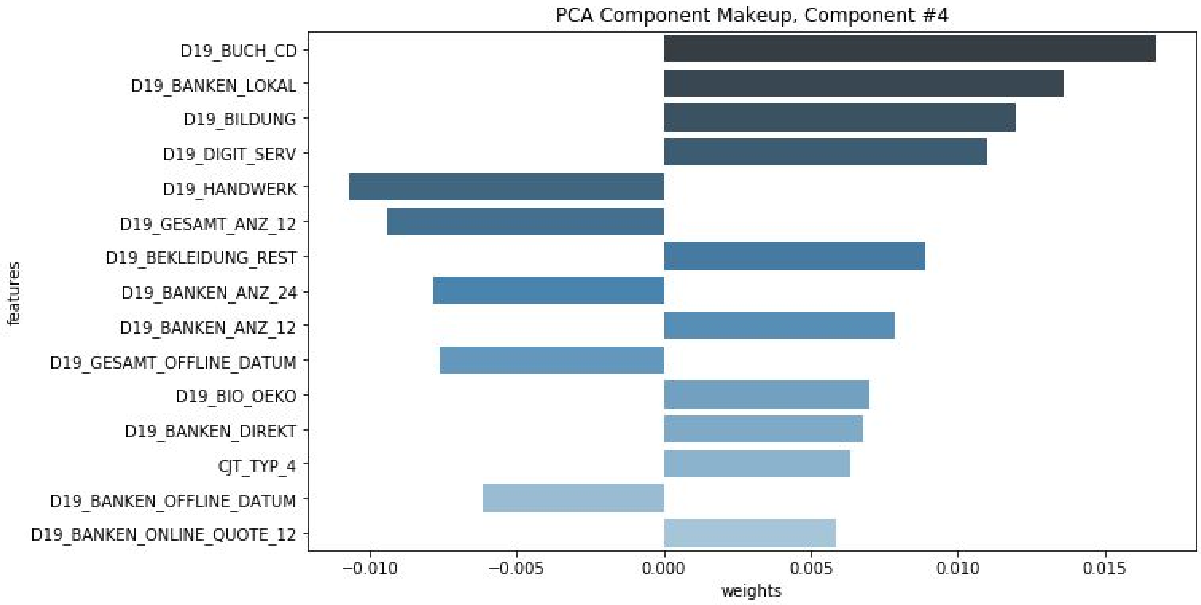
```
Explained Variance is 0.7521851672604789 with 47 components
```

Now, let's take a few looks at the components:



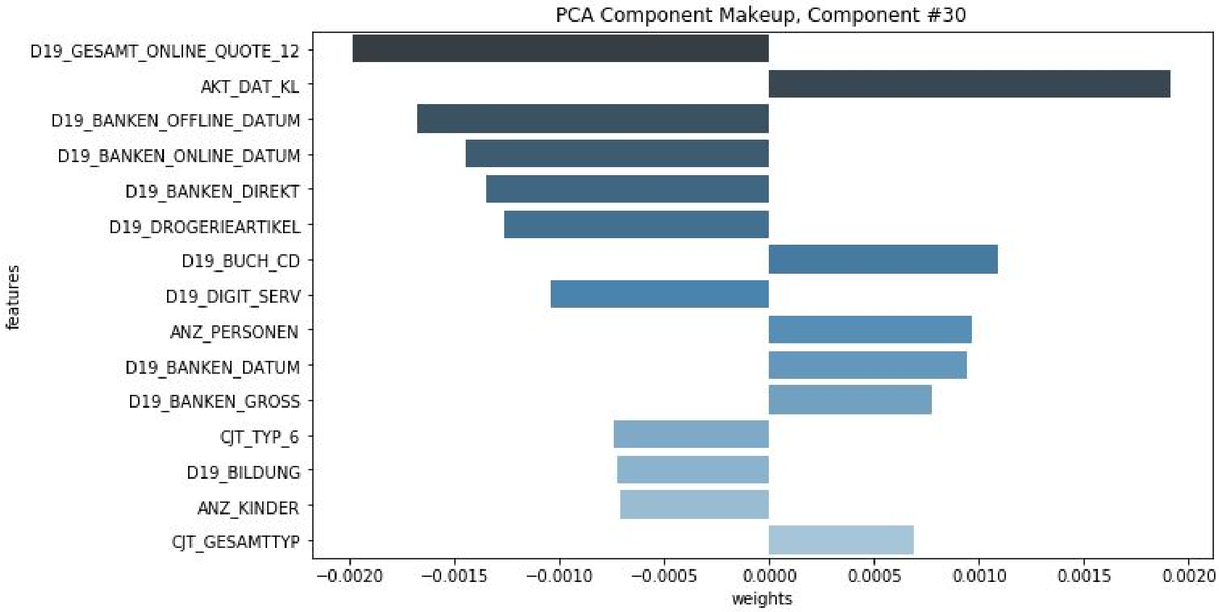
This component is mostly made up of D19 features, meaning mostly transactional activity information. There are some ANZ there which is household information.

D19_GESAMT_DATUM, the biggest component translates to "actuality of the last transaction with the complete file TOTAL"



This component is almost completely made up of transactional activity too, more than the previous one. I can see a lot of "BANKEN" which means it is related to the bank's activity too.

Currently, not all of the column meanings are available to me; but I can see that D19_BUCH_CD is having the biggest impact on this component 14.

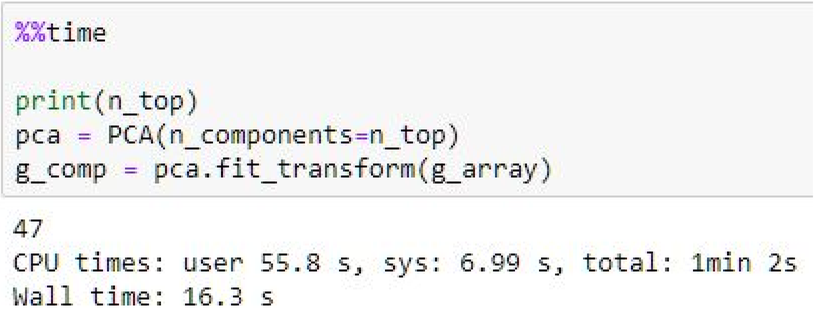


This component is having a big impact from D19_GESAMT_ONLINE_QUOTE_12 and AKT_DAT_KL.

Notice how none of these components are heavily influenced by car data (KBA13), even though most of the columns consisted of car data. This makes one argue that we might not need all these car features after all.

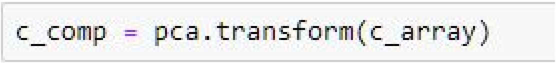
Further on, realistically, we would take these component names and go back to the marketing department or database administrator and have them give us the meanings of each of these feature names so that we know in more detail what exactly these components are made up of.

So, we'll create our pca model again but with 47 components only.



```
%%time

print(n_top)
pca = PCA(n_components=n_top)
g_comp = pca.fit_transform(g_array)
```

```
47
CPU times: user 55.8 s, sys: 6.99 s, total: 1min 2s
Wall time: 16.3 s
```

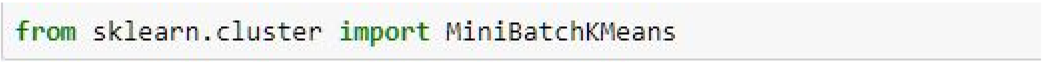


```
c_comp = pca.transform(c_array)
```

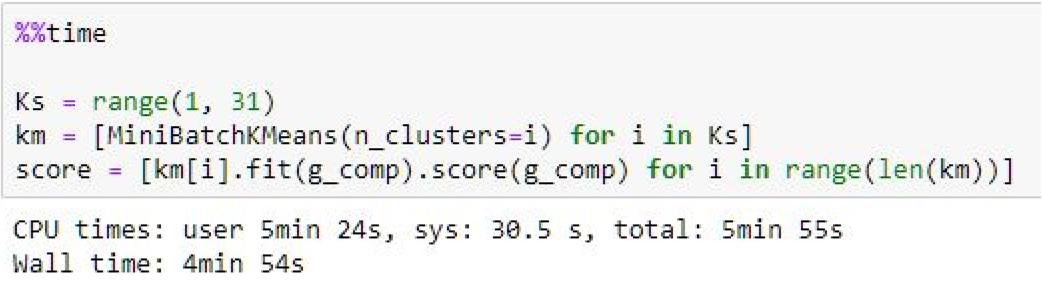
Clustering (K Means)

After trying out Kmeans, it seems to take too long on our large data. So, we will use MiniBatchKmeans instead. This function is known to be much faster than the original, and with a very slight, negligible, decrease in accuracy.

We will try out a wide range of numbers for K, from 1 to 30.



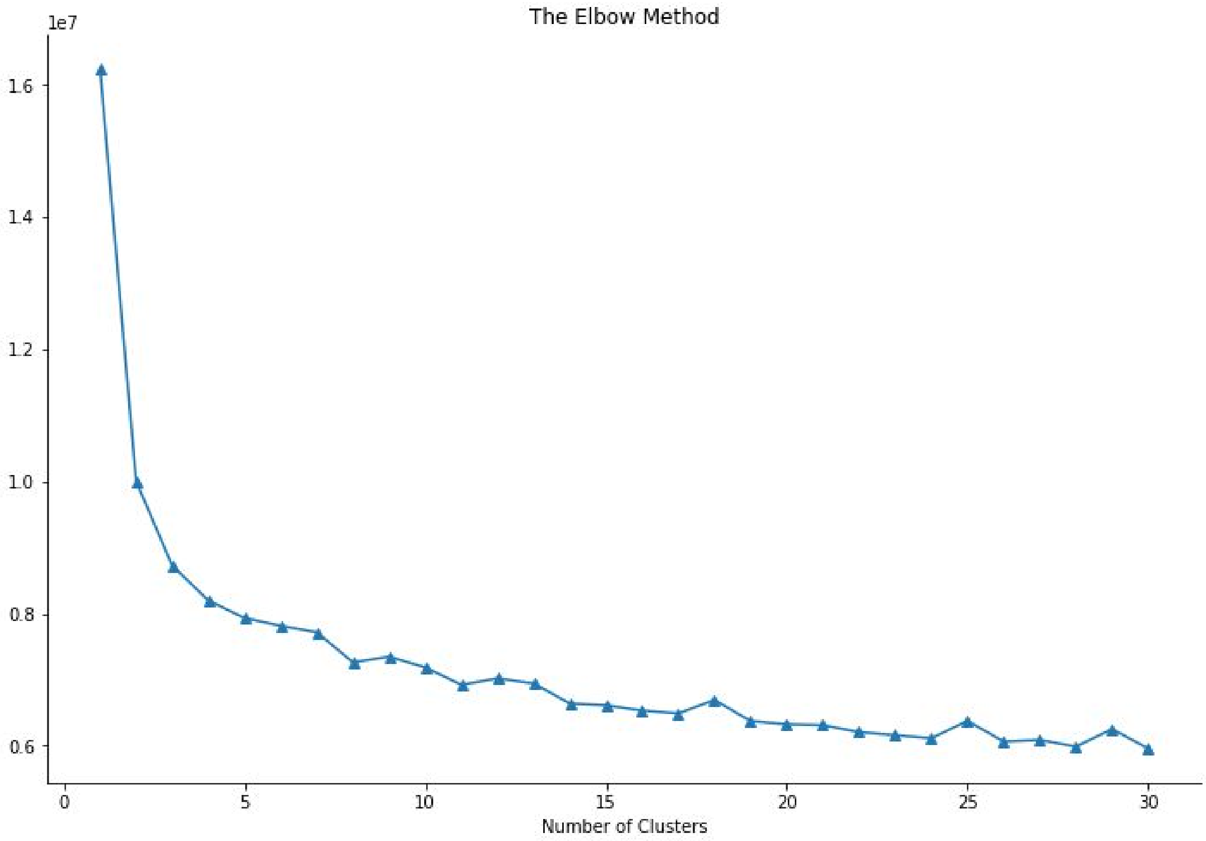
```
from sklearn.cluster import MiniBatchKMeans
```



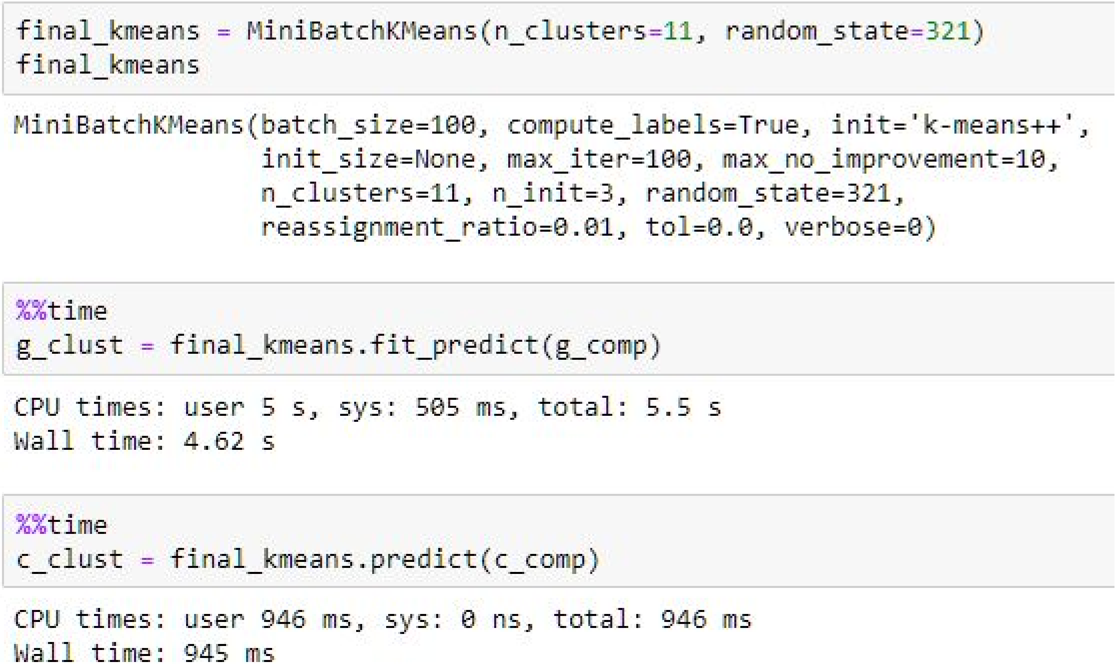
```
%%time

Ks = range(1, 31)
km = [MiniBatchKMeans(n_clusters=i) for i in Ks]
score = [km[i].fit(g_comp).score(g_comp) for i in range(len(km))]
```

```
CPU times: user 5min 24s, sys: 30.5 s, total: 5min 55s
Wall time: 4min 54s
```

Next, we'll plot our Ks to try to find the optimum number using the 'elbow' method:



Looking at this, 8 and 11 seem to be good 'elbow' points. We'll go with 11 so as to have more clusters for our large amount of data.



```
final_kmeans = MiniBatchKMeans(n_clusters=11, random_state=321)
final_kmeans
```

```
MiniBatchKMeans(batch_size=100, compute_labels=True, init='k-means++',
                init_size=None, max_iter=100, max_no_improvement=10,
                n_clusters=11, n_init=3, random_state=321,
                reassignment_ratio=0.01, tol=0.0, verbose=0)
```

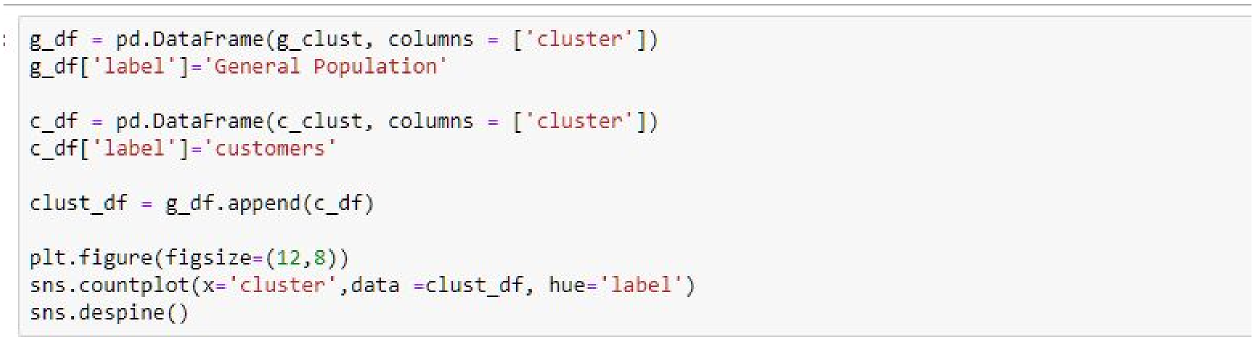
```
%%time
g_clust = final_kmeans.fit_predict(g_comp)
```

```
CPU times: user 5 s, sys: 505 ms, total: 5.5 s
Wall time: 4.62 s
```

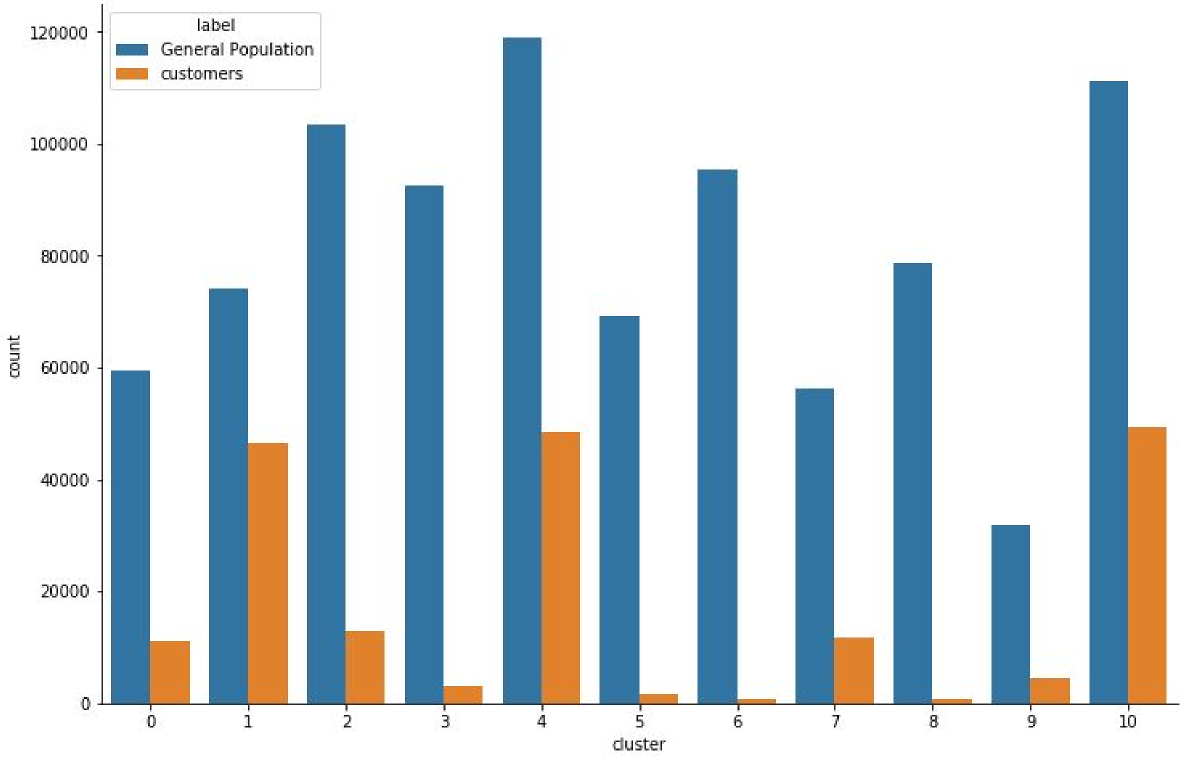
```
%%time
c_clust = final_kmeans.predict(c_comp)
```

```
CPU times: user 946 ms, sys: 0 ns, total: 946 ms
Wall time: 945 ms
```

Now let's see the differences between our datasets for each cluster:

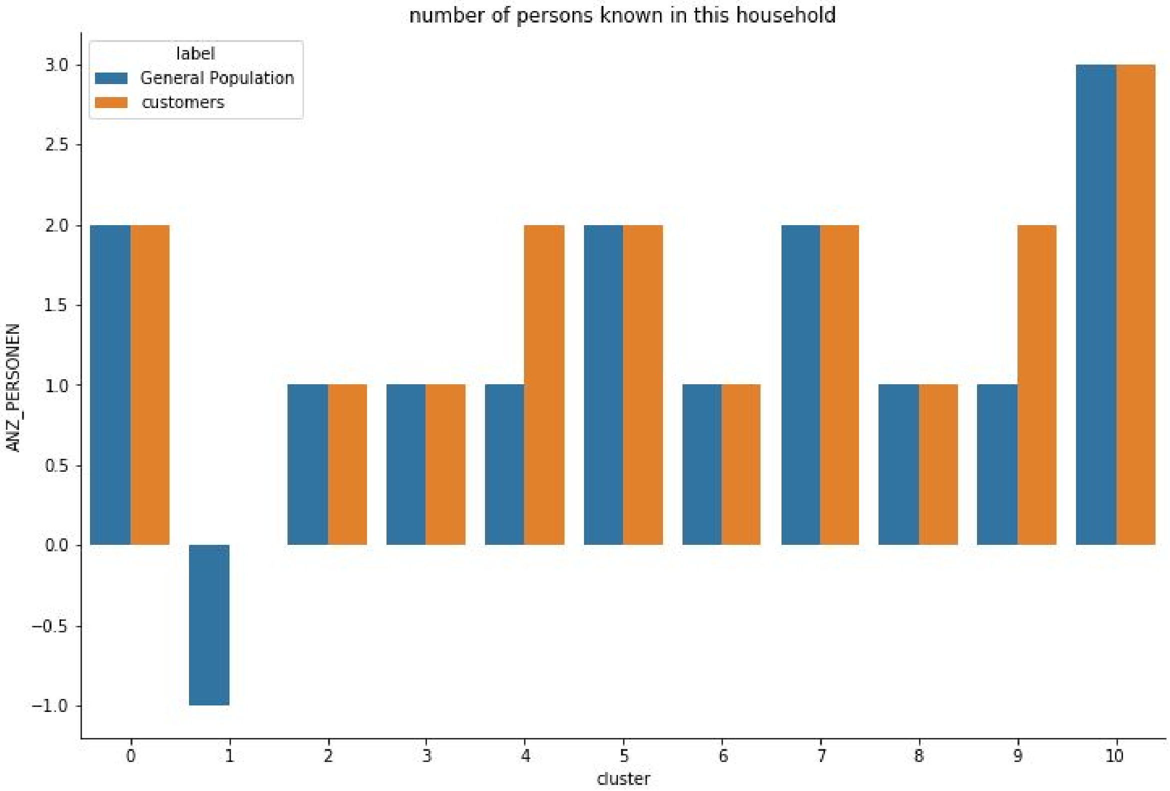


```
g_df = pd.DataFrame(g_clust, columns = ['cluster'])
g_df['label']='General Population'

c_df = pd.DataFrame(c_clust, columns = ['cluster'])
c_df['label']='customers'

clust_df = g_df.append(c_df)

plt.figure(figsize=(12,8))
sns.countplot(x='cluster',data =clust_df, hue='label')
sns.despine()
```

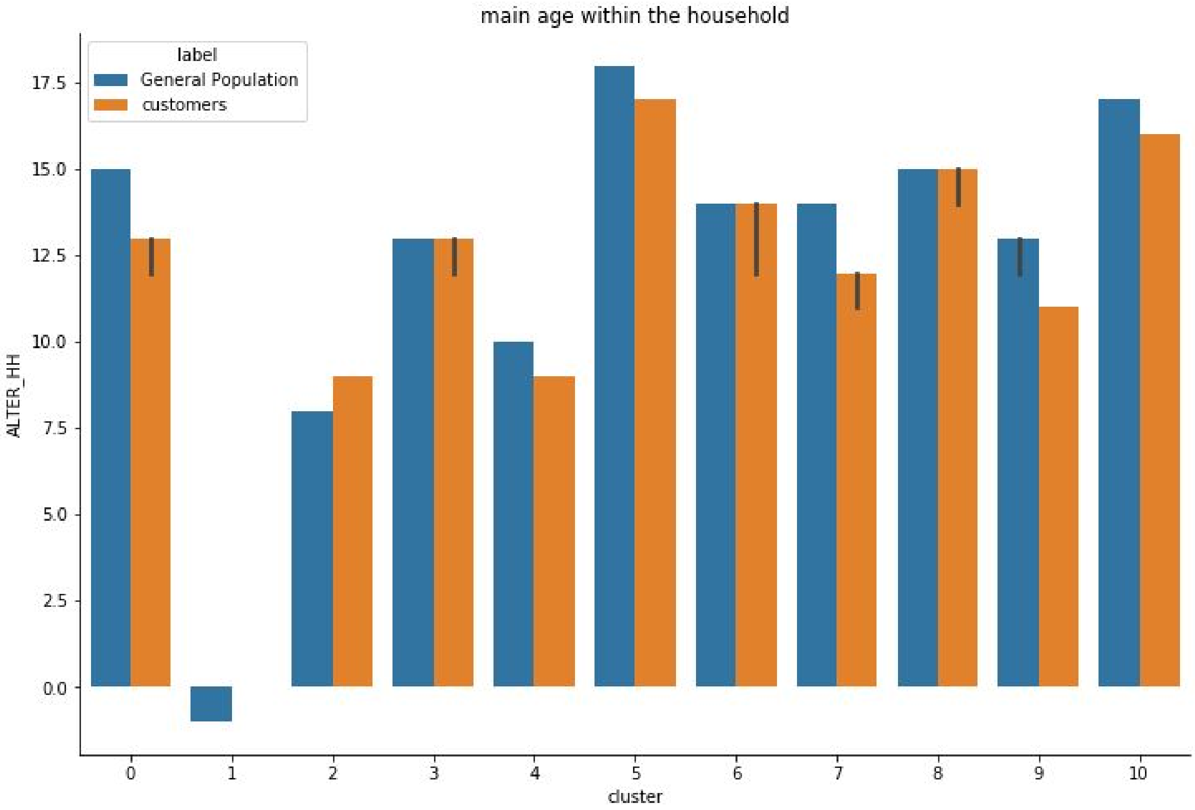


We can see that clusters 4, 10 and 1 are the clusters with most counts for both our General population dataset and our customers dataset.
Interestingly, clusters and 8 and 6 are getting very few counts for customers, yet maintain a good number for the general population.

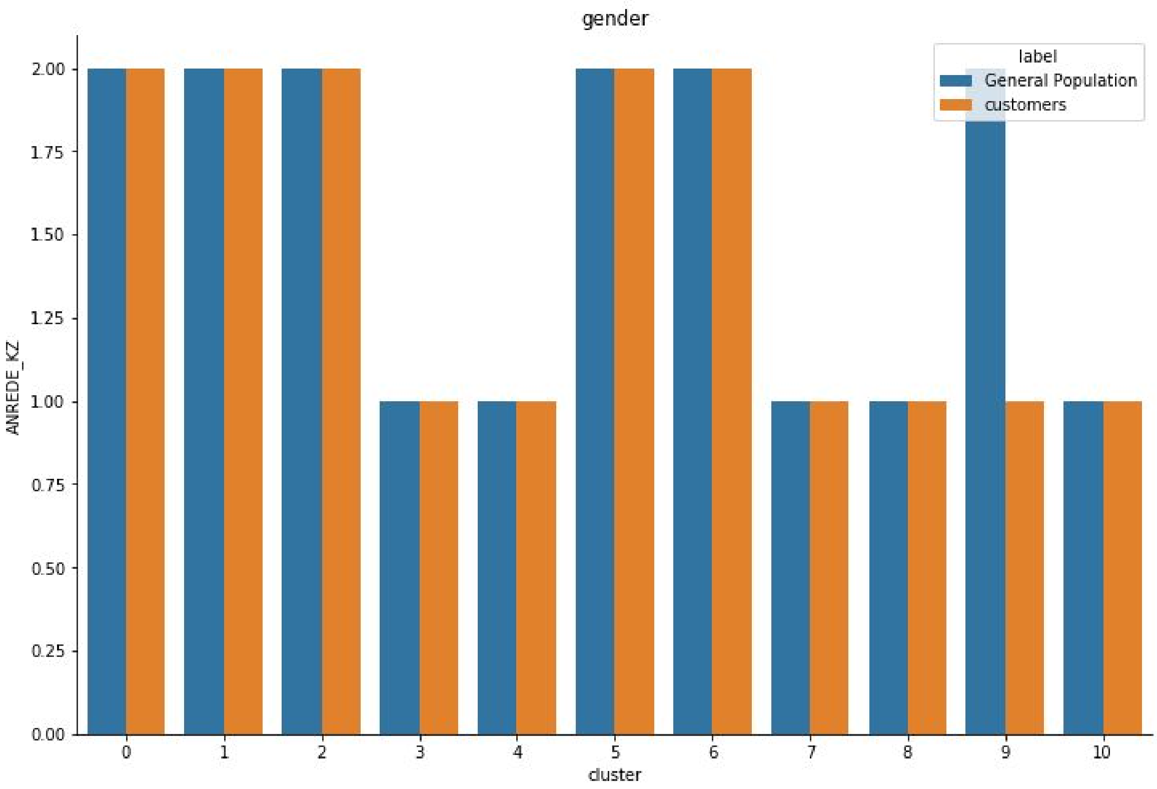
Let's compare a few features and see how they differ between clusters and between our 2 datasets. We will calculate the median for each cluster here to avoid outliers. We won’t use sum, so that we can balance between the 2 datasets:



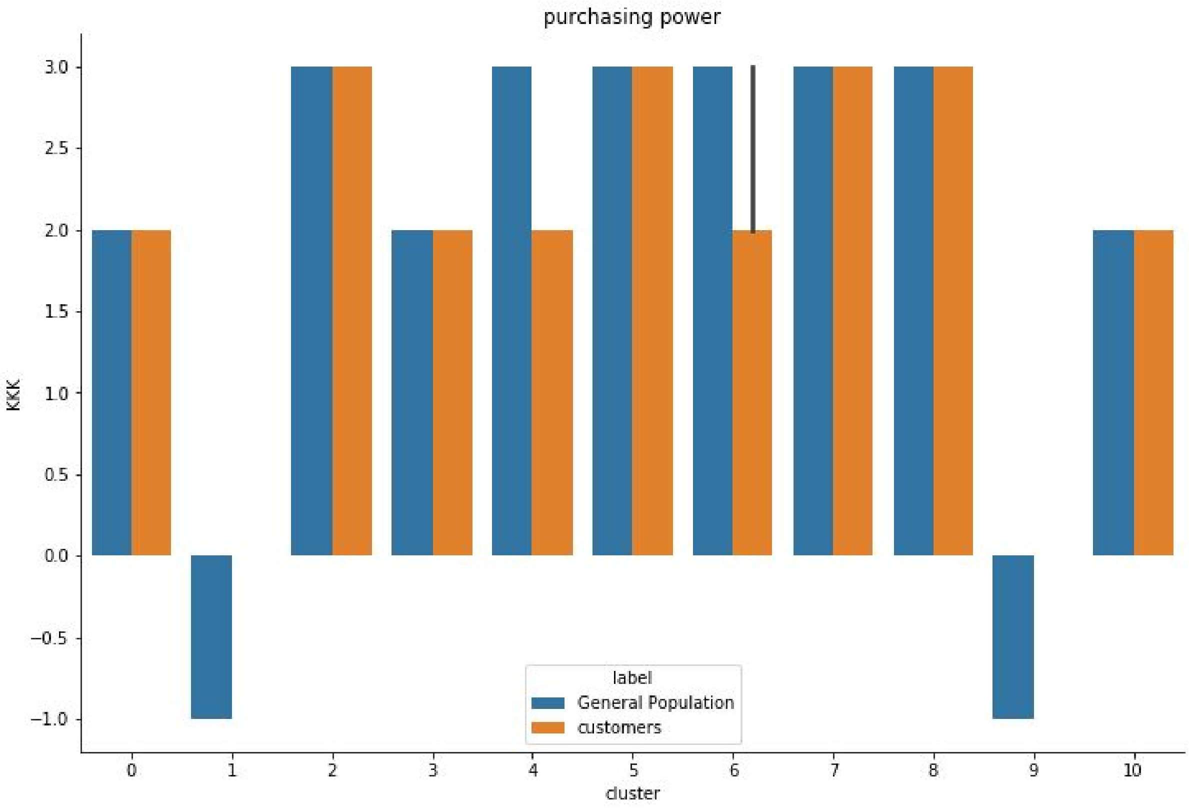
Interestingly, it seems to be either -1 or 0 for cluster 1 for this feature. Otherwise, this feature seems to be favored by customers over the general population for clusters 4 and 9.



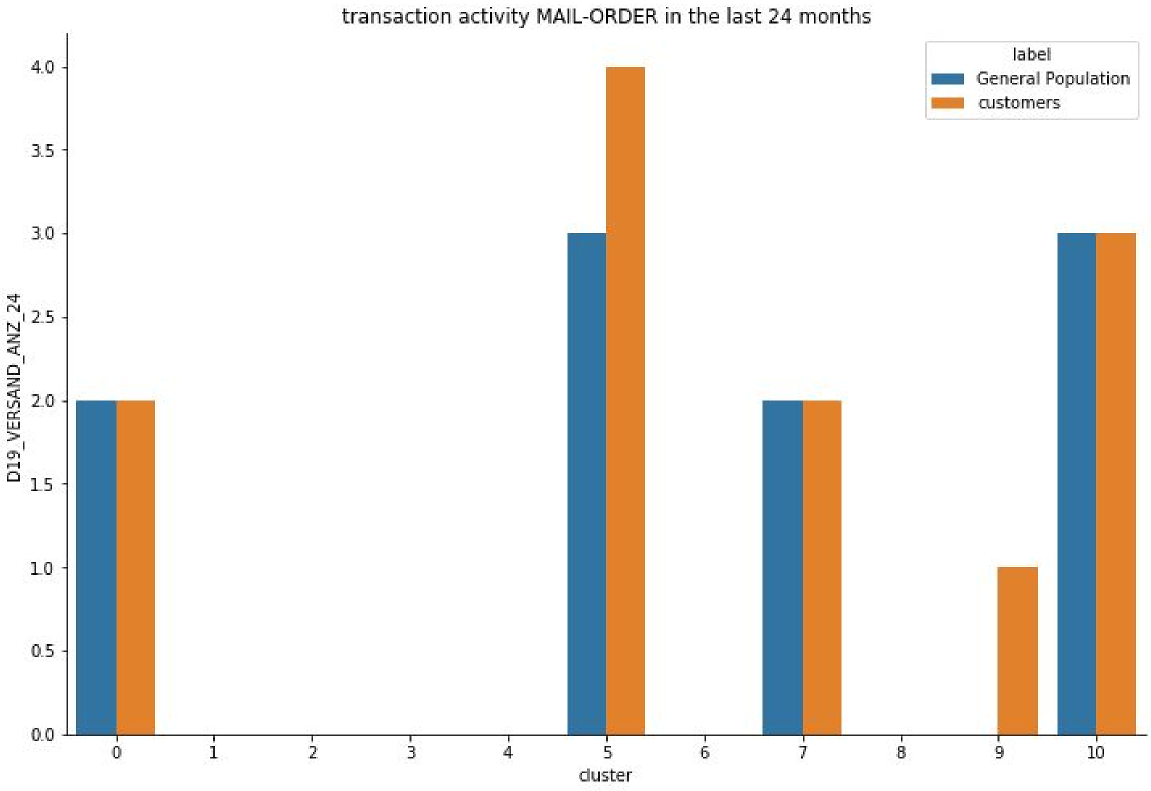
For this feature, cluster 1 is also not favored at all by any of the datasets. The rest are mostly the same, with the general population slightly higher in clusters 0, 4, 5, 7, 9 and 10.



Here we see there are no differences between the 2 datasets, except for cluster 9, where it is widely favored by the general population by about twice as much.

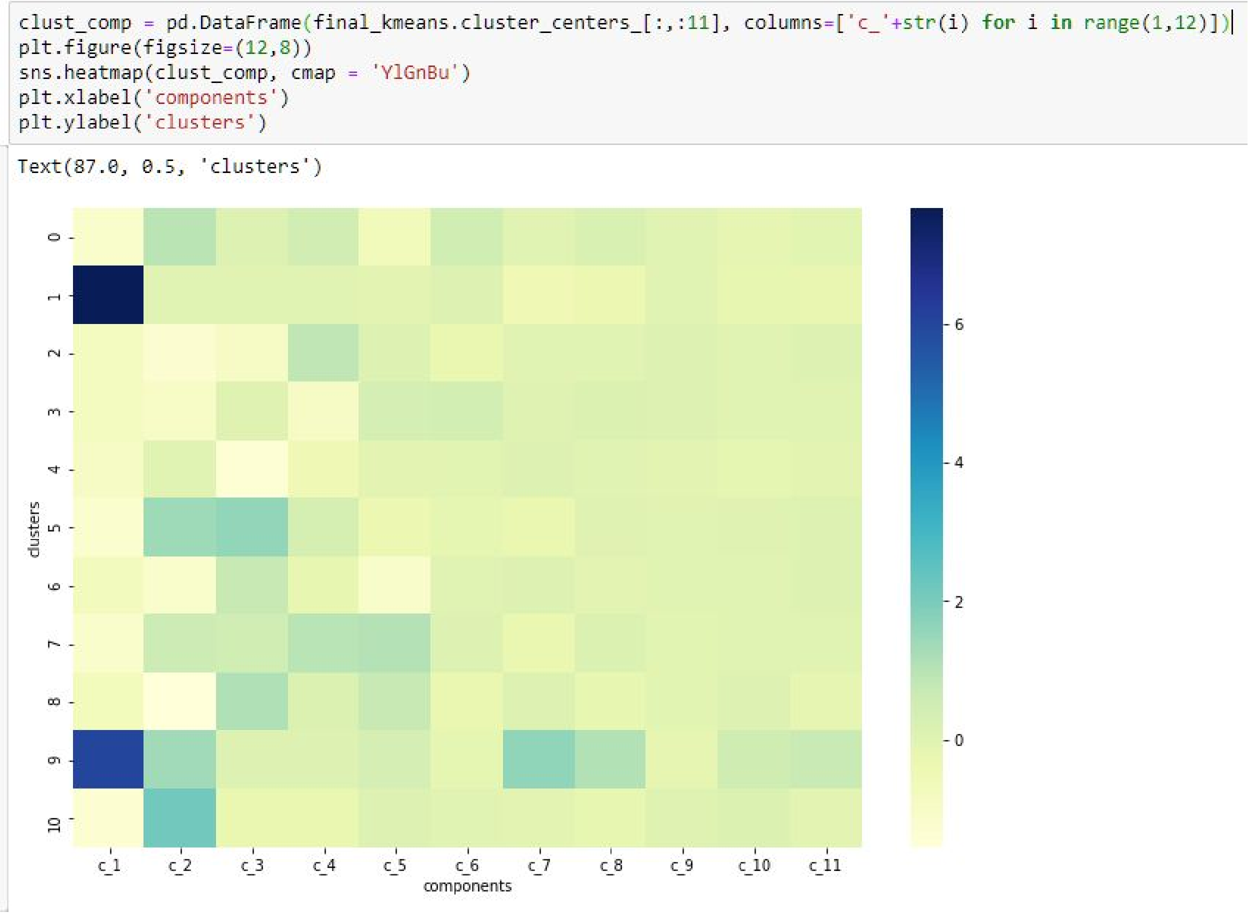


For this we see things are widely different. For starters, we have 0 for customers for clusters 1 and 9, whereas the general population is -1. The other differences are in clusters 4 and 6, where the general population is a ⅓ higher than our customers dataset.



This feature is very interesting in that it is only made up by 5 clusters, with the general population having 0 for cluster 9.

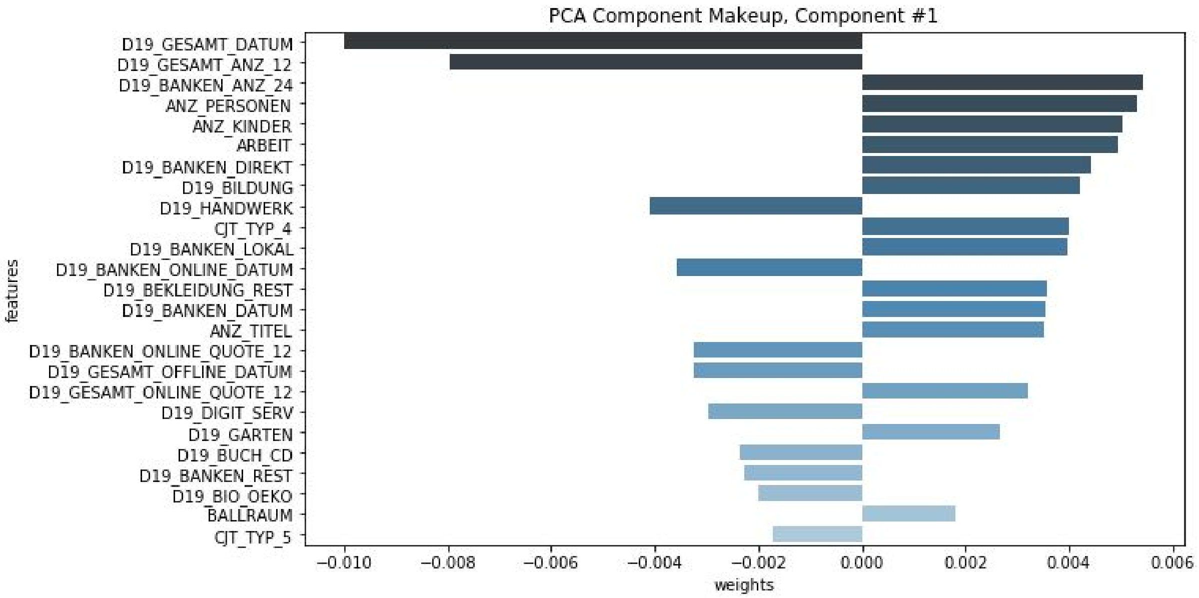
Since we have only 11 clusters, we will see how they compare to our top 11 components:



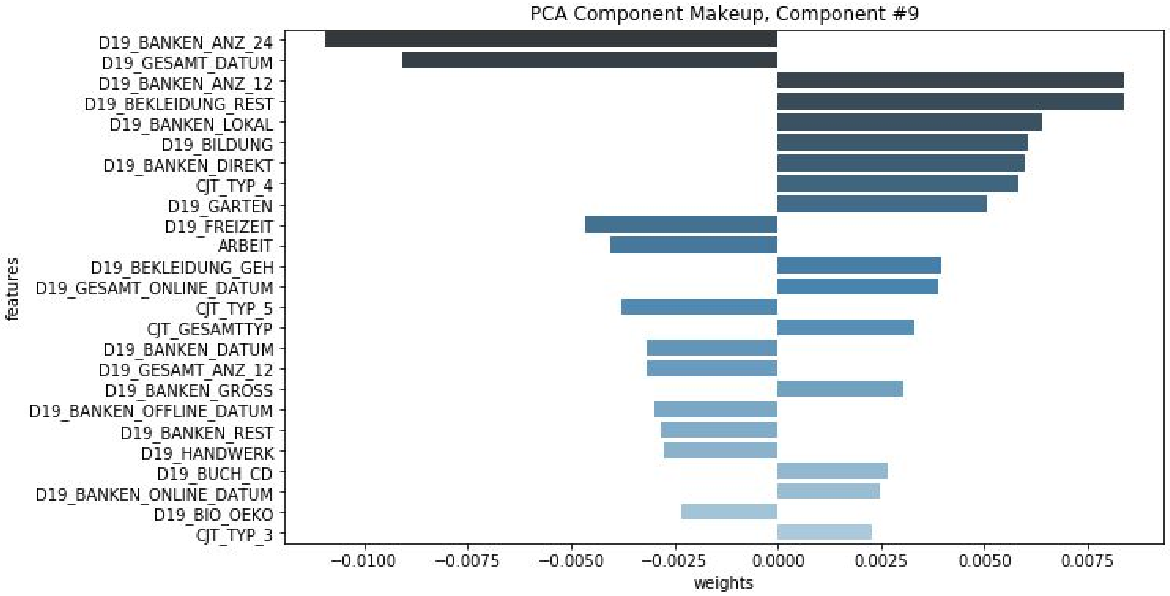
```
clust_comp = pd.DataFrame(final_kmeans.cluster_centers_[:,:11], columns=['c_'+str(i) for i in range(1,12)])
plt.figure(figsize=(12,8))
sns.heatmap(clust_comp, cmap = 'YlGnBu')
plt.xlabel('components')
plt.ylabel('clusters')
```

```
Text(87.0, 0.5, 'clusters')
```

We can see that component 1 is mostly grouped into clusters 9 and 1 here



```
display_component(v,genpop.columns.values, component_num=9, n_weights=25)
```

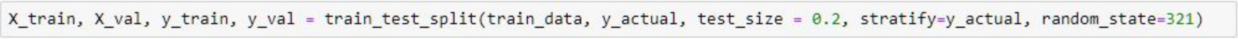


And looking at these it makes sense, since the components look similar.

IV. Supervised Learning: Predictions

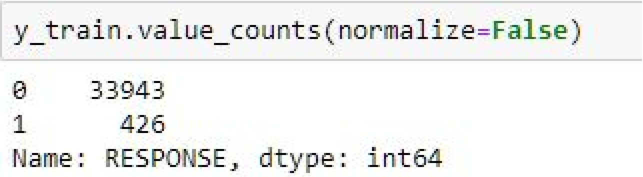
Preparation

Moving on to our predictions, we first need to do a few steps. Since our test dataset does not contain any response column, we will have to create a validation set to use for evaluating our model as it tries to improve itself.



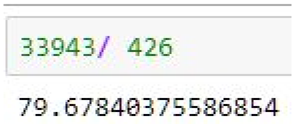
```
X_train, X_val, y_train, y_val = train_test_split(train_data, y_actual, test_size = 0.2, stratify=y_actual, random_state=321)
```

We need to make sure that we are using the ‘stratify’ argument to keep the same ratio of 0 and 1 response.



```
y_train.value_counts(normalize=False)
```

```
0    33943
1      426
Name: RESPONSE, dtype: int64
```

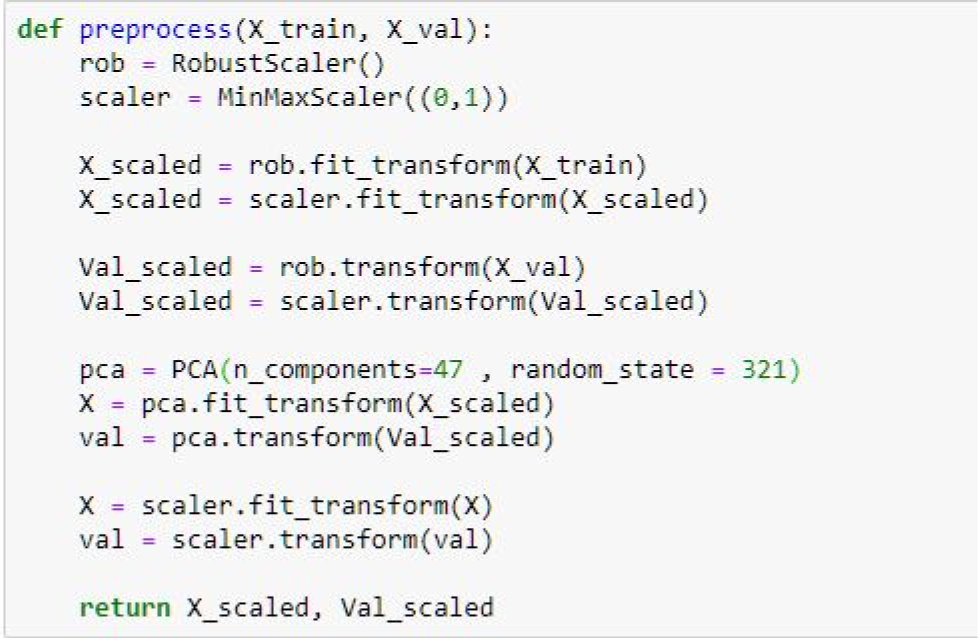


```
33943/ 426
```

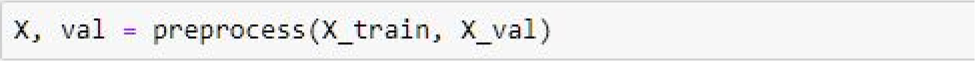
```
79.67840375586854
```

Taking a quick look at our target variable, we see that the data is heavily imbalanced in favor of 0 for response. The ratio seems to be 80:1 for 0:1 Response. We will need to take this into account for all our prediction algorithms.

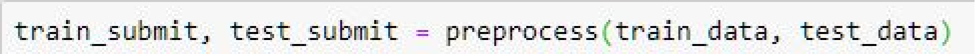
Moving on, we will use the same preprocessing steps we did for Kmeans, but this time it’s all rapped up neatly into 1 function:



```
def preprocess(X_train, X_val):
    rob = RobustScaler()
    scaler = MinMaxScaler((0,1))

    X_scaled = rob.fit_transform(X_train)
    X_scaled = scaler.fit_transform(X_scaled)

    Val_scaled = rob.transform(X_val)
    Val_scaled = scaler.transform(Val_scaled)

    pca = PCA(n_components=47 , random_state = 321)
    X = pca.fit_transform(X_scaled)
    val = pca.transform(Val_scaled)

    X = scaler.fit_transform(X)
    val = scaler.transform(val)

    return X_scaled, Val_scaled
```



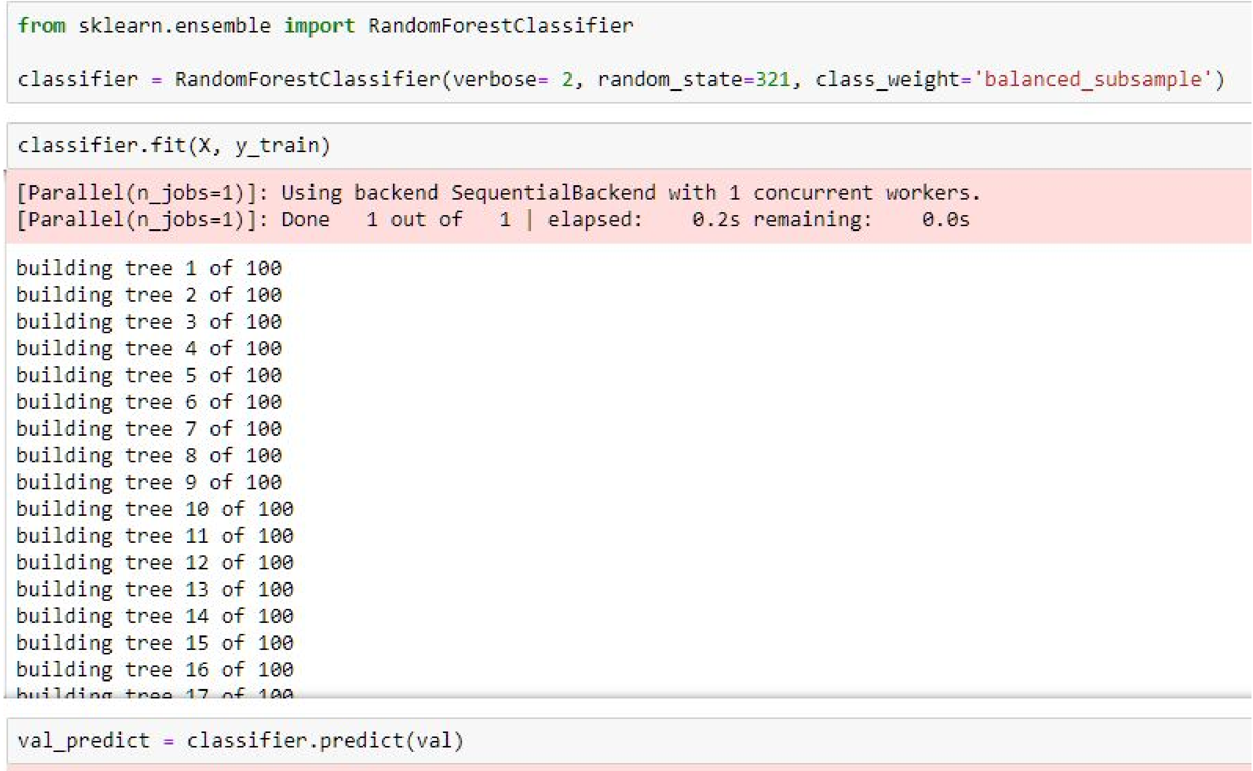
```
X, val = preprocess(X_train, X_val)
```



```
train_submit, test_submit = preprocess(train_data, test_data)
```

Basic Model

Now we’re ready to construct a basic model to get some base performance values.



```
from sklearn.ensemble import RandomForestClassifier

classifier = RandomForestClassifier(verbose= 2, random_state=321, class_weight='balanced_subsample')
```

```
classifier.fit(X, y_train)
```

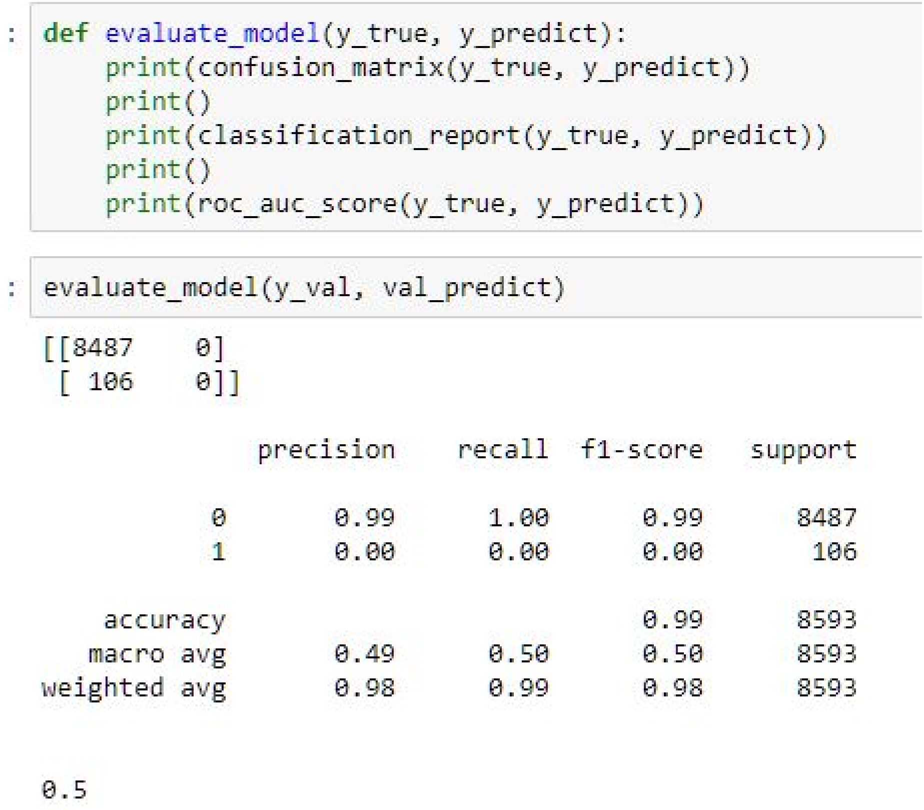
```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.2s remaining:    0.0s
```

```
building tree 1 of 100
building tree 2 of 100
building tree 3 of 100
building tree 4 of 100
building tree 5 of 100
building tree 6 of 100
building tree 7 of 100
building tree 8 of 100
building tree 9 of 100
building tree 10 of 100
building tree 11 of 100
building tree 12 of 100
building tree 13 of 100
building tree 14 of 100
building tree 15 of 100
building tree 16 of 100
building tree 17 of 100
```

```
val_predict = classifier.predict(val)
```

We use RandomForestClassifier for its known reliability and robustness. We used ‘balanced_subsample’ for the class_weight to have the classifier try to take into account our target imbalances.

Then, we’ll create an evaluation function that calculates the confusion matrix, classification report that consists of precision, recall and f1-score values, and finally displays the ROC-AUC score.



```
def evaluate_model(y_true, y_predict):
    print(confusion_matrix(y_true, y_predict))
    print()
    print(classification_report(y_true, y_predict))
    print()
    print(roc_auc_score(y_true, y_predict))
```

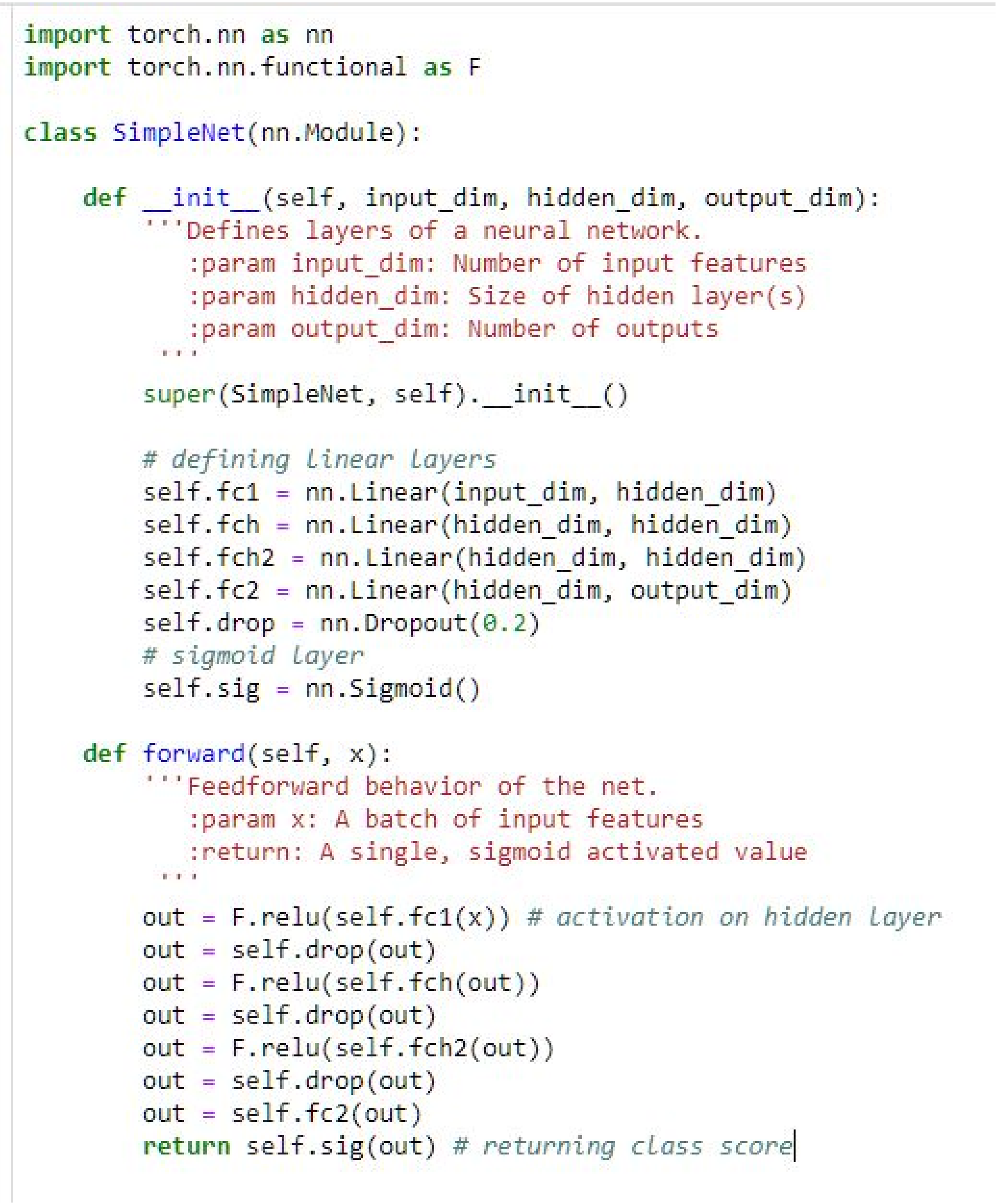
```
evaluate_model(y_val, val_predict)
```

```
[[8487    0]
 [ 106    0]]

              precision    recall  f1-score   support

           0       0.99      1.00      0.99      8487
           1       0.00      0.00      0.00       106

    accuracy                           0.99      8593
   macro avg       0.49      0.50      0.50      8593
weighted avg       0.98      0.99      0.98      8593


0.5
```

Well, even with the balanced setting for class_weight, a basic RandomForest completely fails to predict any positives for our dataset.

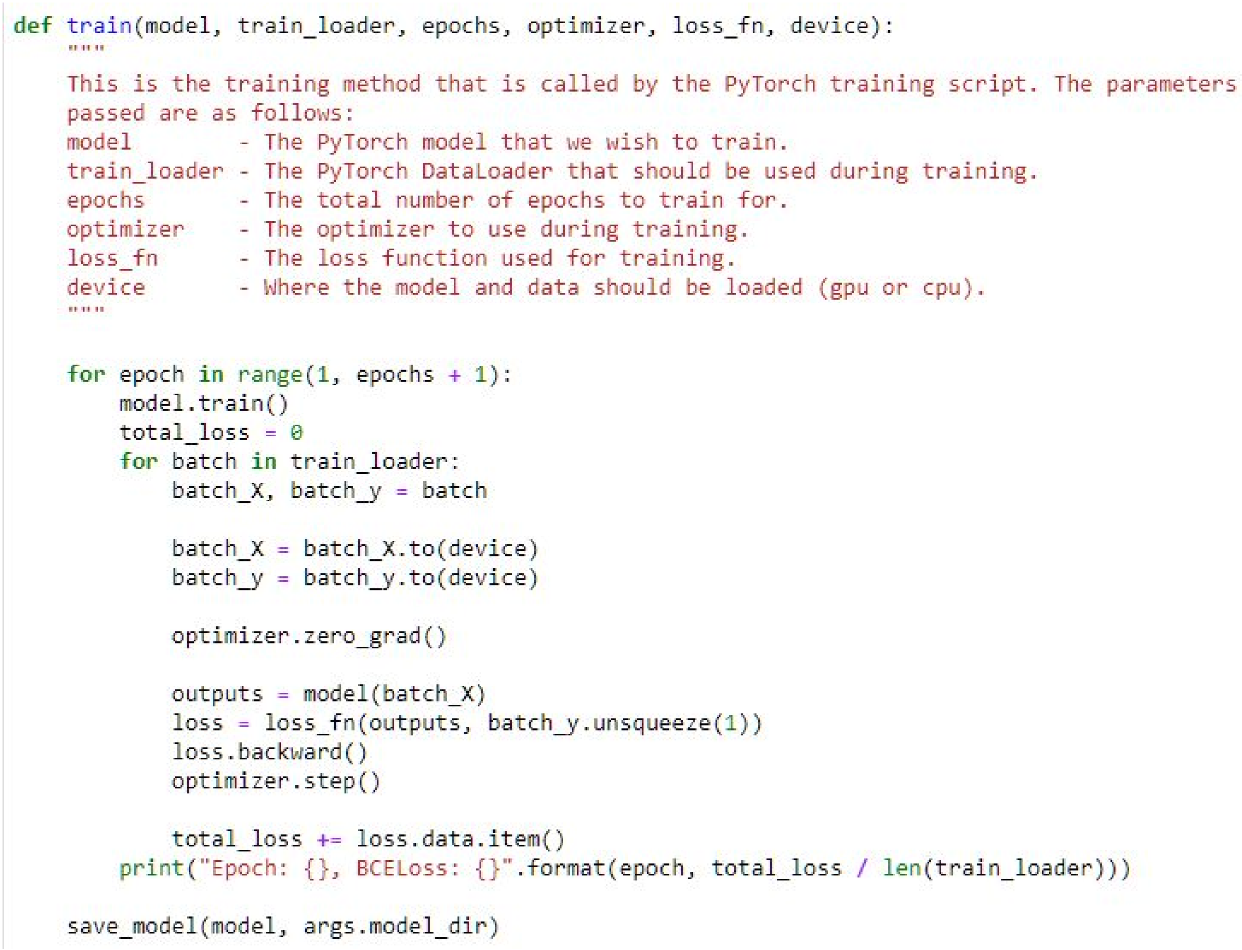
Neural Network

We will try out Pytorch’s Neural Network (NN) algorithm next to see if it can perform better.



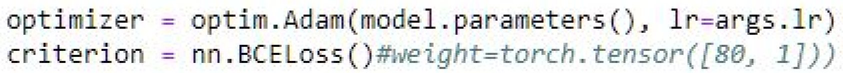
```
import torch.nn as nn
import torch.nn.functional as F

class SimpleNet(nn.Module):

    def __init__(self, input_dim, hidden_dim, output_dim):
        '''Defines layers of a neural network.
           :param input_dim: Number of input features
           :param hidden_dim: Size of hidden layer(s)
           :param output_dim: Number of outputs
         '''
        super(SimpleNet, self).__init__()

        # defining linear layers
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fch = nn.Linear(hidden_dim, hidden_dim)
        self.fch2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.drop = nn.Dropout(0.2)
        # sigmoid layer
        self.sig = nn.Sigmoid()

    def forward(self, x):
        '''Feedforward behavior of the net.
           :param x: A batch of input features
           :return: A single, sigmoid activated value
         '''
        out = F.relu(self.fc1(x)) # activation on hidden layer
        out = self.drop(out)
        out = F.relu(self.fch(out))
        out = self.drop(out)
        out = F.relu(self.fch2(out))
        out = self.drop(out)
        out = self.fc2(out)
        return self.sig(out) # returning class score
```

This NN network consists of an input layer, 2 hidden layers and an output layer. After each layer, we use a dropout of 0.2 to help avoid any overfitting. Finally, we use the Sigmoid function to produce our probabilities between 0 and 1.



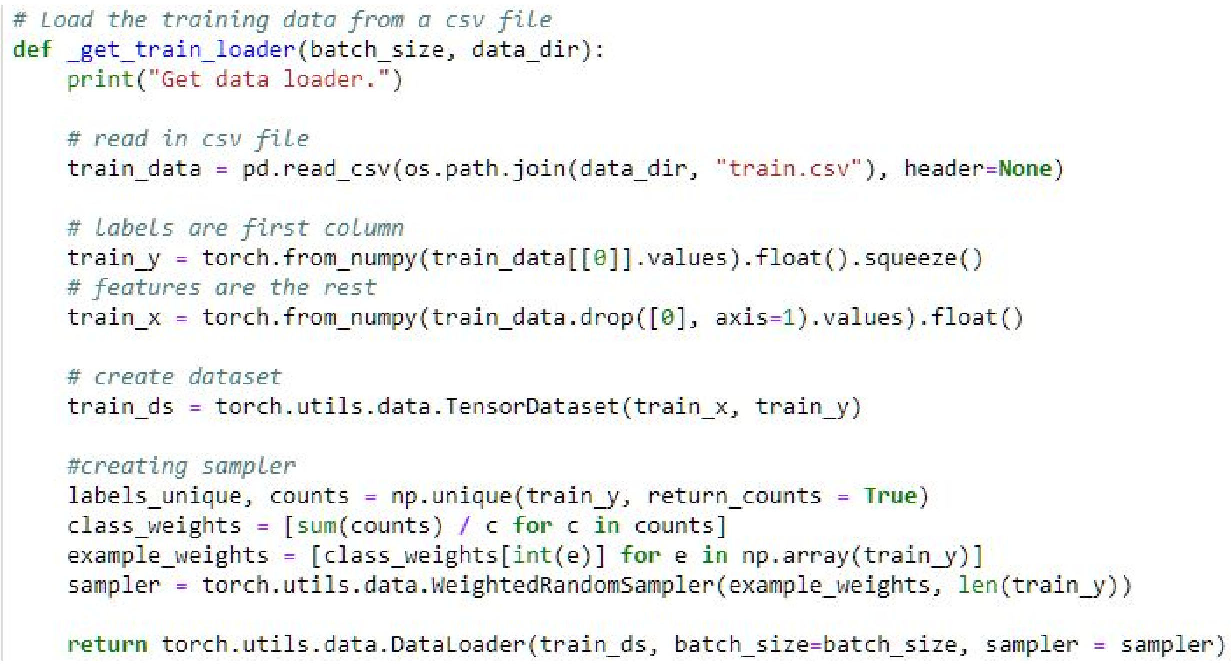
```python
def train(model, train_loader, epochs, optimizer, loss_fn, device):
    """
    This is the training method that is called by the PyTorch training script. The parameters
    passed are as follows:
    model        - The PyTorch model that we wish to train.
    train_loader - The PyTorch DataLoader that should be used during training.
    epochs       - The total number of epochs to train for.
    optimizer    - The optimizer to use during training.
    loss_fn      - The loss function used for training.
    device       - Where the model and data should be loaded (gpu or cpu).
    """

    for epoch in range(1, epochs + 1):
        model.train()
        total_loss = 0
        for batch in train_loader:
            batch_X, batch_y = batch

            batch_X = batch_X.to(device)
            batch_y = batch_y.to(device)

            optimizer.zero_grad()

            outputs = model(batch_X)
            loss = loss_fn(outputs, batch_y.unsqueeze(1))
            loss.backward()
            optimizer.step()

            total_loss += loss.data.item()
        print("Epoch: {}, BCELoss: {}".format(epoch, total_loss / len(train_loader)))

    save_model(model, args.model_dir)
```

We use a generic train function that follows the same procedure of backpropagation formula.



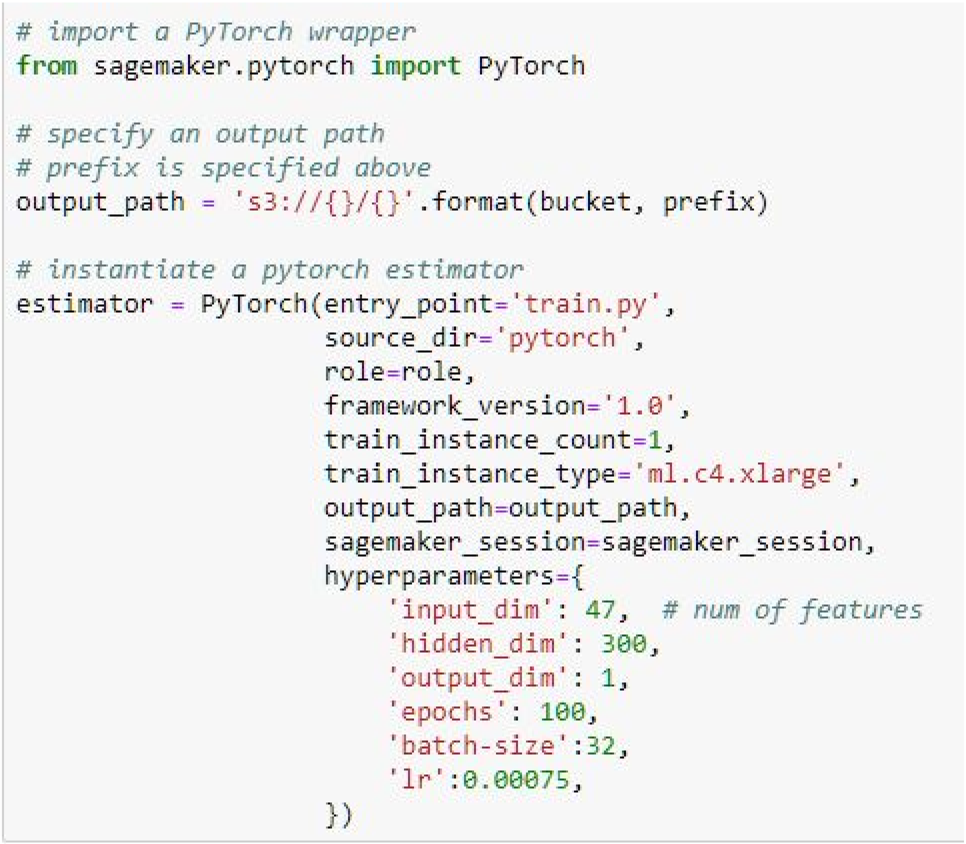
```python
optimizer = optim.Adam(model.parameters(), lr=args.lr)
criterion = nn.BCELoss()#weight=torch.tensor([80, 1]))
```

We'll use Adam for its robustness and reliability, and BCELoss, which stands for Binary Cross-Entropy loss, since we're dealing with a single binary target variable.



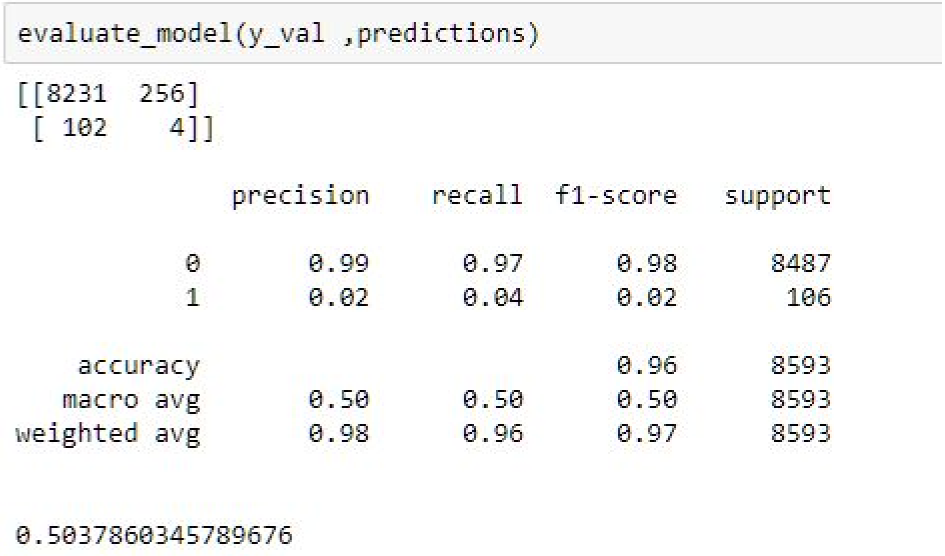
```
# Load the training data from a csv file
def _get_train_loader(batch_size, data_dir):
    print("Get data loader.")

    # read in csv file
    train_data = pd.read_csv(os.path.join(data_dir, "train.csv"), header=None)

    # labels are first column
    train_y = torch.from_numpy(train_data[[0]].values).float().squeeze()
    # features are the rest
    train_x = torch.from_numpy(train_data.drop([0], axis=1).values).float()

    # create dataset
    train_ds = torch.utils.data.TensorDataset(train_x, train_y)

    #creating sampler
    labels_unique, counts = np.unique(train_y, return_counts = True)
    class_weights = [sum(counts) / c for c in counts]
    example_weights = [class_weights[int(e)] for e in np.array(train_y)]
    sampler = torch.utils.data.WeightedRandomSampler(example_weights, len(train_y))

    return torch.utils.data.DataLoader(train_ds, batch_size=batch_size, sampler = sampler)
```

An important note we need to make here is that we used pytorch’s WeightedRandomSampler to help with the imbalance. The important section is the part after ‘creating sampler’. This is a method that uses the over-sampling technique.



```
# import a PyTorch wrapper
from sagemaker.pytorch import PyTorch

# specify an output path
# prefix is specified above
output_path = 's3://{}/{}'.format(bucket, prefix)

# instantiate a pytorch estimator
estimator = PyTorch(entry_point='train.py',
                    source_dir='pytorch',
                    role=role,
                    framework_version='1.0',
                    train_instance_count=1,
                    train_instance_type='ml.c4.xlarge',
                    output_path=output_path,
                    sagemaker_session=sagemaker_session,
                    hyperparameters={
                        'input_dim': 47,  # num of features
                        'hidden_dim': 300,
                        'output_dim': 1,
                        'epochs': 100,
                        'batch-size':32,
                        'lr':0.00075,
                    })
```

We’ll use these hyperparameters and pass it to Sagemaker.
When we’re done we try again to see how it does on our validation set:



```
evaluate_model(y_val ,predictions)
```

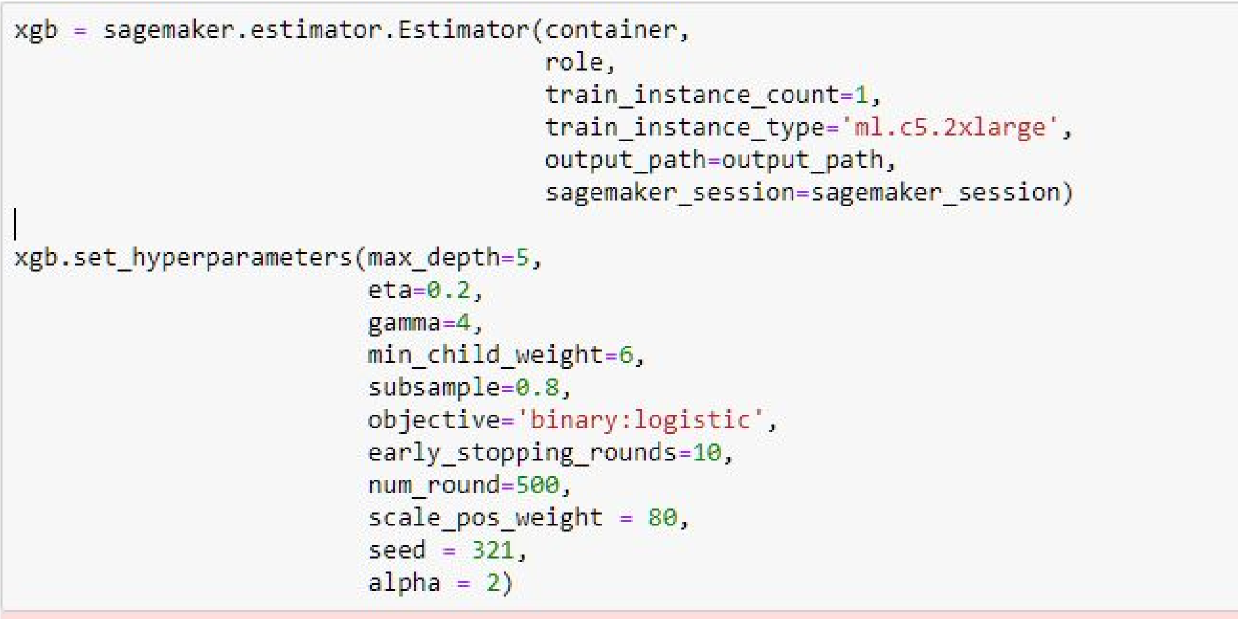
```
[[8231  256]
 [ 102    4]]

              precision    recall  f1-score   support

           0       0.99      0.97      0.98      8487
           1       0.02      0.04      0.02       106

    accuracy                           0.96      8593
   macro avg       0.50      0.50      0.50      8593
weighted avg       0.98      0.96      0.97      8593


0.5037860345789676
```

It seems to be only slightly better than the base model, even after we did our best to try to balance the weights. One good point is that at least it's not predicting all 0s anymore.

Optimization by XGBoost

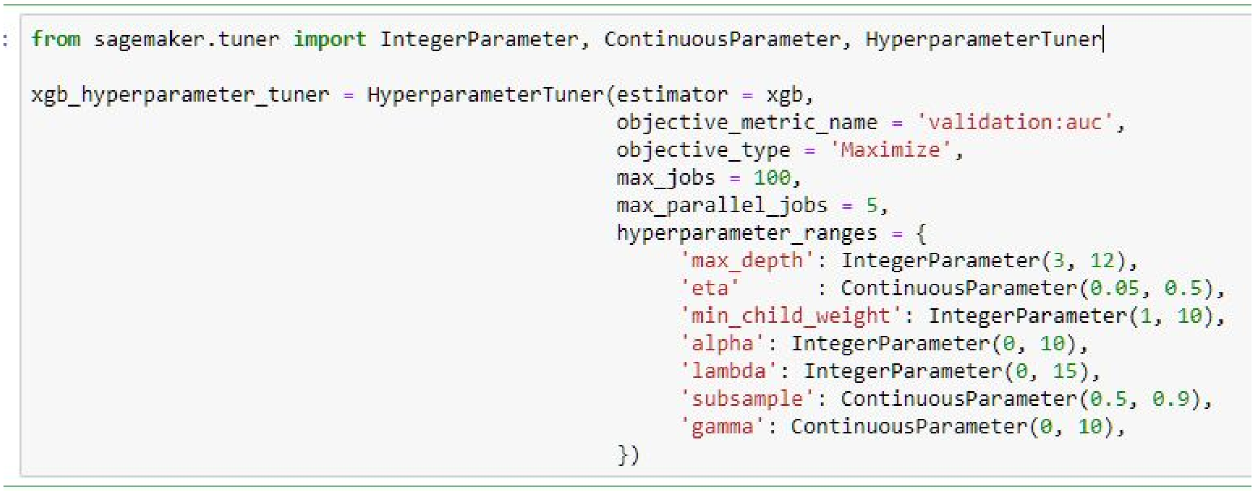
Our final solution to tackle this problem would be through hyperparameter optimization. I tried as best I could to use pytorch for optimization through Sagemaker, but the lack of documentation or information proved too difficult. For that reason, we will move on to try one of Sagemaker's built-in algorithms, XGBoost.

XGBoost (XGB) is a very reliable and, again, robust algorithm that is widely used in Kaggle competitions. It is also generally much faster than NN with the ability to still be complex enough when the hyperparameters are properly tuned.



```
xgb = sagemaker.estimator.Estimator(container,
                                    role,
                                    train_instance_count=1,
                                    train_instance_type='ml.c5.2xlarge',
                                    output_path=output_path,
                                    sagemaker_session=sagemaker_session)

xgb.set_hyperparameters(max_depth=5,
                        eta=0.2,
                        gamma=4,
                        min_child_weight=6,
                        subsample=0.8,
                        objective='binary:logistic',
                        early_stopping_rounds=10,
                        num_round=500,
                        scale_pos_weight = 80,
                        seed = 321,
                        alpha = 2)
```

These base hyperparameters we will set as default before optimization. We will use early_stopping_rounds of 10 to stop our model early if it does not improve its loss after 10 rounds at any point through our 500 (num_)rounds. Since our model is imbalanced at a ratio of 80:1, we will use the scale_pos_weight argument and set it to 80.

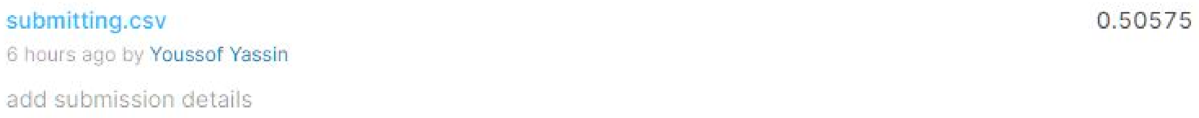


```
from sagemaker.tuner import IntegerParameter, ContinuousParameter, HyperparameterTuner

xgb_hyperparameter_tuner = HyperparameterTuner(estimator = xgb,
                                               objective_metric_name = 'validation:auc',
                                               objective_type = 'Maximize',
                                               max_jobs = 100,
                                               max_parallel_jobs = 5,
                                               hyperparameter_ranges = {
                                                    'max_depth': IntegerParameter(3, 12),
                                                    'eta'      : ContinuousParameter(0.05, 0.5),
                                                    'min_child_weight': IntegerParameter(1, 10),
                                                    'alpha': IntegerParameter(0, 10),
                                                    'lambda': IntegerParameter(0, 15),
                                                    'subsample': ContinuousParameter(0.5, 0.9),
                                                    'gamma': ContinuousParameter(0, 10),
                                               })
```

Then, we set up Sagemaker's optimization algorithm: HyperparameterTuner. We will use validation:auc as our objective_metric, as well as making sure to maximize it as much as possible, while setting 100 max_jobs, to be run in parallel, 5 jobs at a time.

Note that one parameter we did not set is the type of search HyperparameterTuner uses. This is because by default, the algorithm uses Bayesian Search. This is what we want, since searching randomly would take too much time and would prove too costly.

We run this model and send our submission over to Kaggle.

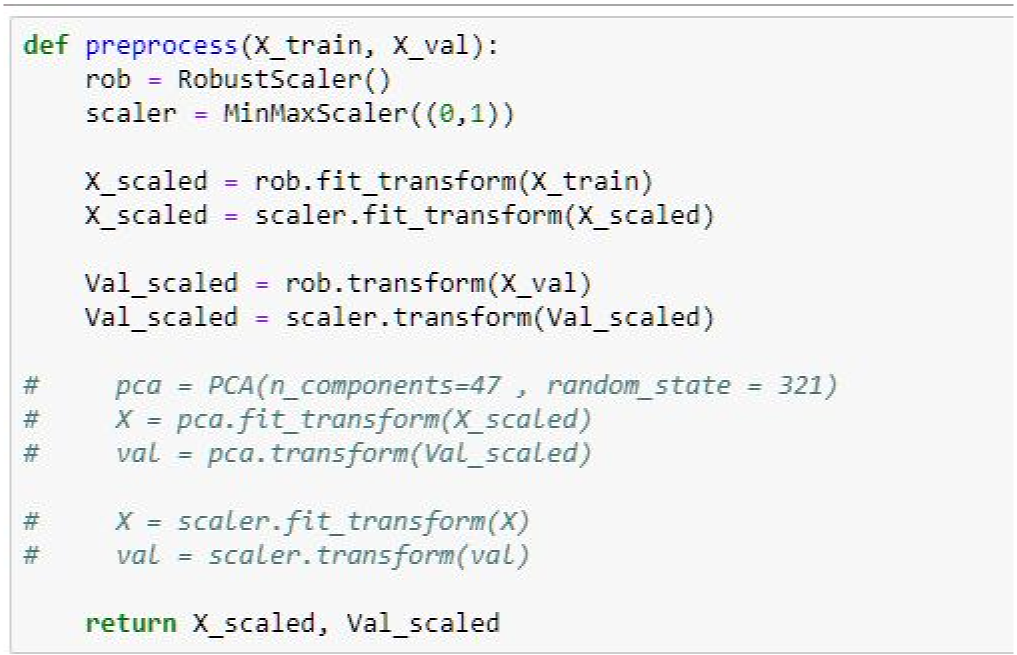


submitting.csv
6 hours ago by Youssof Yassin

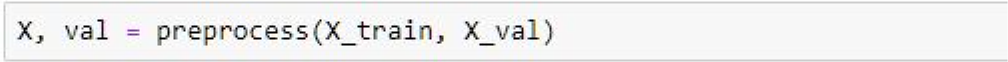
add submission details

0.50575

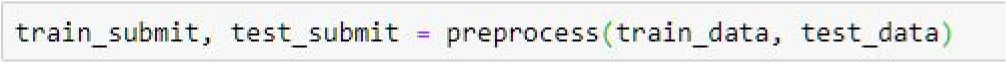
Our Auc score is still stuck at about 0.5. Something seems fundamentally wrong here, if we are failing with 3 different models, and after optimizing so heavily.
On a hunch, we will try removing the PCA process we are doing to our training sets:



```
def preprocess(X_train, X_val):
    rob = RobustScaler()
    scaler = MinMaxScaler((0,1))

    X_scaled = rob.fit_transform(X_train)
    X_scaled = scaler.fit_transform(X_scaled)

    Val_scaled = rob.transform(X_val)
    Val_scaled = scaler.transform(Val_scaled)

#     pca = PCA(n_components=47 , random_state = 321)
#     X = pca.fit_transform(X_scaled)
#     val = pca.transform(Val_scaled)

#     X = scaler.fit_transform(X)
#     val = scaler.transform(val)

    return X_scaled, Val_scaled
```

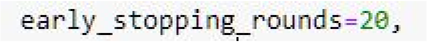


```
X, val = preprocess(X_train, X_val)
```

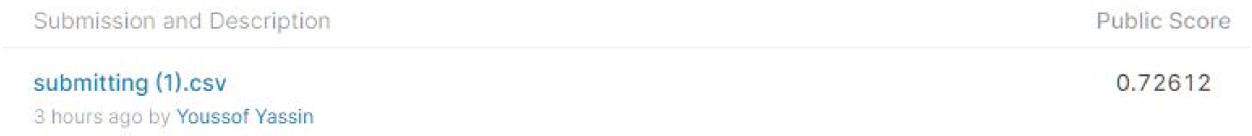


```
train_submit, test_submit = preprocess(train_data, test_data)
```

Then, we will mostly keep everything else the same and increase the early_stopping_rounds a bit:



```
early_stopping_rounds=20,
```



Submission and Description	Public Score
submitting (1).csv 3 hours ago by Youssof Yassin	0.72612

Finally, we have achieved a proper score!

V. Conclusion

Reflection

This was a very difficult project, initially because the data was so large. We had to upgrade from the default 2 core 4 GiB Sagemaker notebook instance to a 8 core 32 GiB one to even begin to load the data to start working.

Another major challenge was figuring out which features were important and which were not. Even with the provided informational datasets, it was still too difficult to understand what most of the columns meant. Ideally, when this is done as part of a job, we would go back to someone knowledgeable and have them help us understand what to keep and what should not be used. For example, why were there so many car data columns, even though they were eventually not important features, according to our PCA algorithm?

We did our best in preprocessing by removing features with empty values more than 20%. We dealt with categorical features by using the most frequent value and changed each one into dummy variables (one for each unique value). Then, we dealt with numerical features by simple using the median. Finally, we used a RobustScaler followed by MinMaxScaler to finish up the preprocessing.

For the clustering, we first performed PCA to reduce our data to few components, that made up about 75% total explained variance. We used a MiniBatchKmeans algorithm and found the appropriate K using the elbow method. Finally, we compared the differences for each cluster between the 2 major datasets, customers and general population.

For machine learning predictions, we used 3 models. The first was a RandomForest classifier that used the class_weight option to try to fix the imbalances that were happening at a ratio of 80:1. We then moved on to a pytorch Neural Network, since it's known to usually do well with big datasets. After failing to get a good ROC-AUC score, we moved to our final model, XGBoost. This one was tuned using Sagemaker's HyperparameterTuner which uses a Bayesian Search method to optimize over a range of hyperparameters.

After getting a bad score on Kaggle, we did one last trick, which is to remove the PCA section from our preprocess function and just use all the clean, but unreduced, features. This finally got us a proper score of 0.726.

Now, while it's a bit depressing that we did not reach a 0.8 score like those at the top of the Kaggle leaderboard, it's still so much better than the 0.5 score that we kept getting. Thinking about it, using this model in production for sending out advertisements, or mail orders, is still much better than just giving them out randomly to the general population. So, I personally, consider it a success.

Improvement

Of course, a lot more work can be done to improve the model. After finding out in the very end that the PCA step was causing problems for our machine learning predictions, we can later go back to our pytorch NN or even the RandomForest to see how they do with this step removed.

If it's within budget, we can try XGBoost optimization again, but this time by having 1000 or more jobs running, instead of just 100. We do not know for sure if XGBoost reached its maximum potential, so running the optimization for much longer might prove beneficial.

Maybe later if Amazon provides better documentation or tutorials, we can use our pytorch model for optimization, instead of XGBoost, to see how it compares and if it can do any better.

Finally, one last attempt at improvement could be concentrated at the preprocessing part of the project. Particularly, we might need to work in much more detail with the features to truly understand which features are redundant and throwing off our metrics. If we understand the features better, we might even come up with ways where we can combine 2 features together or create new features entirely.

Citation:

The following github repositories were very helpful and inspired some of the code used for this project.

https://github.com/udacity/ML_SageMaker_Studies

https://github.com/udacity/sagemaker-deployment