Self Case Study - 1

Customer Relationship Prediction - Appetency

In [ ]:

```
!pip install -U scikit-learn
```

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.7/dist-packages (0.24.2)
Requirement already satisfied: numpy>=1.13.3 in /usr/local/lib/python3.7/dist-packages (from scikit-lea
rn) (1.19.5)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from sci
kit-learn) (2.2.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit-lear
n) (1.0.1)
Requirement already satisfied: scipy>=0.19.1 in /usr/local/lib/python3.7/dist-packages (from scikit-lea
rn) (1.4.1)
```

In [ ]:

```
!pip install dython
```

```
Requirement already satisfied: dython in /usr/local/lib/python3.7/dist-packages (0.6.7)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from dython) (1.4.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from dython) (1.19.5)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from dython) (3.2.
2)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.7/dist-packages (from dython) (0.
24.2)
Requirement already satisfied: scikit-plot>=0.3.7 in /usr/local/lib/python3.7/dist-packages (from dytho
n) (0.3.7)
Requirement already satisfied: seaborn in /usr/local/lib/python3.7/dist-packages (from dython) (0.11.1)
Requirement already satisfied: pandas>=0.23.4 in /usr/local/lib/python3.7/dist-packages (from dython) (
1.1.5)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-packages (from pandas>=0.2
3.4->dython) (2018.9)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/dist-packages (from p
andas>=0.23.4->dython) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil
>=2.7.3->pandas>=0.23.4->dython) (1.15.0)
Requirement already satisfied: joblib>=0.10 in /usr/local/lib/python3.7/dist-packages (from scikit-plot
>=0.3.7->dython) (1.0.1)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dis
t-packages (from matplotlib->dython) (2.4.7)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplo
tlib->dython) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib-
>dython) (0.10.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from sci
kit-learn->dython) (2.2.0)
```

In [ ]:

```
!pip install fast-ml
```

```
Requirement already satisfied: fast-ml in /usr/local/lib/python3.7/dist-packages (3.68)
```

In [ ]:

```
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
```

```python
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn.preprocessing import OrdinalEncoder
from sklearn.manifold import TSNE
from sklearn.preprocessing import PolynomialFeatures

import missingno as msno
from dython import nominal

import pickle

from sklearn.cluster import DBSCAN, KMeans

from fast_ml.utilities import display_all
from fast_ml.feature_selection import get_duplicate_features

from sklearn.model_selection import train_test_split

from prettytable import PrettyTable

%matplotlib inline
```

Loading data

In [ ]:

```python
data = pd.read_csv('orange_small_train.data', sep = '\t')
```

In [ ]:

```python
data.shape
```

Out[ ]:

```
(50000, 230)
```

In [ ]:

```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Columns: 230 entries, Var1 to Var230
dtypes: float64(191), int64(1), object(38)
memory usage: 87.7+ MB
```

There are total of 50k datapoints and each datapoint has 230 features.

In [ ]:

```python
data.head()
```

Out[ ]:

	Var1	Var2	Var3	Var4	Var5	Var6	Var7	Var8	Var9	Var10	Var11	Var12	Var13	Var14	Var15	Var16	Var17	Var18
0	NaN	NaN	NaN	NaN	NaN	1526.0	7.0	NaN	NaN	NaN	NaN	NaN	184.0	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	525.0	0.0	NaN	NaN	NaN	NaN	NaN	0.0	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	5236.0	7.0	NaN	NaN	NaN	NaN	NaN	904.0	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	0.0	NaN	NaN	NaN	NaN	NaN	0.0	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	1029.0	7.0	NaN	NaN	NaN	NaN	NaN	3216.0	NaN	NaN	NaN	NaN	NaN

5 rows × 230 columns

In [ ]:

```
data.describe()
```

Out[ ]:

	Var1	Var2	Var3	Var4	Var5	Var6	Var7	Var8	
count	702.000000	1241.000000	1240.000000	1579.000000	1.487000e+03	44471.000000	44461.000000	0.0	702.000
mean	11.487179	0.004029	425.298387	0.125396	2.387933e+05	1326.437116	6.809496	NaN	48.1452
std	40.709951	0.141933	4270.193518	1.275481	6.441259e+05	2685.693668	6.326053	NaN	154.777
min	0.000000	0.000000	0.000000	0.000000	0.000000e+00	0.000000	0.000000	NaN	0.00000
25%	0.000000	0.000000	0.000000	0.000000	0.000000e+00	518.000000	0.000000	NaN	4.00000
50%	0.000000	0.000000	0.000000	0.000000	0.000000e+00	861.000000	7.000000	NaN	20.0000
75%	16.000000	0.000000	0.000000	0.000000	1.187425e+05	1428.000000	7.000000	NaN	46.0000
max	680.000000	5.000000	130668.000000	27.000000	6.048550e+06	131761.000000	140.000000	NaN	2300.00

8 rows × 192 columns

In [ ]:

```
appetency_labels = pd.read_csv('orange_small_train_appetency.labels', header = None,names = ['Appetency
'])
```

In [ ]:

```
appetency_labels.head()
```

Out[ ]:

	Appetency
0	-1
1	-1
2	-1
3	-1
4	-1

In [ ]:

```
appetency_labels['Appetency'] = appetency_labels.Appetency.apply(lambda x: 0 if (x == -1) else x)
```

In [ ]:

```
appetency_labels.shape
```

Out[ ]:

```
(50000, 1)
```

Splitting data into train and test before data analysis

In [ ]:

```
X_train_appetency, X_test_appetency, y_train_appetency, y_test_appetency = train_test_split(data,appete
```

```
ncy_labels, test_size = 0.2, stratify = appetency_labels)
```

EDA

Class distribution

In [ ]:

```
def plot_class_dist(x, data):
    sns.countplot(x = x,data =  data)
    plt.title('{} class label value counts'.format(x))
    plt.show()
```

Appetency

In [ ]:

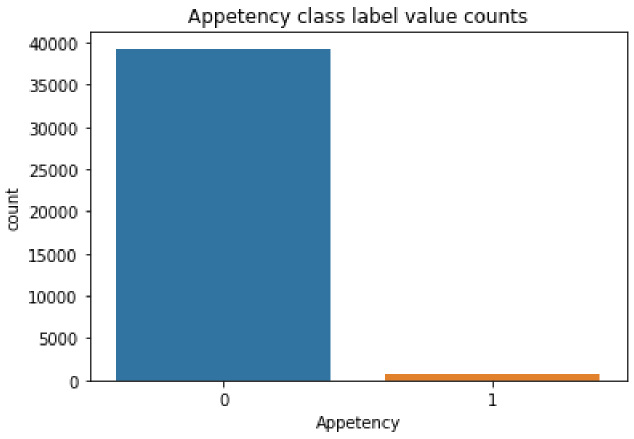
```
y_train_appetency.value_counts()
```

Out[ ]:

```
Appetency
0            39288
1              712
dtype: int64
```

In [ ]:

```
plot_class_dist('Appetency', y_train_appetency)
```



Observation:

- Data w.r.t appetency label is highly imbalanced.

Counting total NaNs for each feature

In [ ]:

```
print('Number of features which only have NaNs present: ',(X_train_appetency.isna().sum() == X_train_ap
petency.shape[0]).sum())
```

```
Number of features which only have NaNs present:  18
```

In [ ]:

```
all_nan_columns = np.array(X_train_appetency.columns[X_train_appetency.isna().sum() == X_train_appetenc
y.shape[0]])
```

In [ ]:

```
print('Number of features which do not countain NaNs:',(X_train_appetency.notna().sum() == X_train_appe
tency.shape[0]).sum())
```

```
Number of features which do not countain NaNs: 19
```

In [ ]:

```
not_nan_columns = np.array(X_train_appetency.columns[X_train_appetency.notna().sum() == X_train_appeten
cy.shape[0]])
```

In [ ]:

```
not_nan_columns
```

Out[ ]:

```
array(['Var57', 'Var73', 'Var113', 'Var193', 'Var195', 'Var196', 'Var198',
       'Var204', 'Var207', 'Var210', 'Var211', 'Var212', 'Var216',
       'Var220', 'Var221', 'Var222', 'Var226', 'Var227', 'Var228'],
      dtype=object)
```

In [ ]:

```
X_train_appetency.dtypes[not_nan_columns]
```

Out[ ]:

```
Var57     float64
Var73       int64
Var113    float64
Var193     object
Var195     object
Var196     object
Var198     object
Var204     object
Var207     object
Var210     object
Var211     object
Var212     object
Var216     object
Var220     object
Var221     object
Var222     object
Var226     object
Var227     object
Var228     object
dtype: object
```

Observation:

    - Out of 19 columns which do not have any missing data, 3 are numerical and 16 are categorical.

In [ ]:

```
#https://stackoverflow.com/questions/26266362/how-to-count-the-nan-values-in-a-column-in-pandas-datafra
me
nan_count_array = []
for i in X_train_appetency.columns:
    nan_count = X_train_appetency[i].isna().sum()
    nan_count_array.append(nan_count)
```

In [ ]:

```
#to-do: tabular form
x = PrettyTable()
x.add_column('Features', list(X_train_appetency.columns))
x.add_column('Number of NaNs',nan_count_array)
```

In [ ]:

```
print(x)
```

```
+----------+----------------+
| Features | Number of NaNs |
+----------+----------------+
|   Var1   |     39427      |
|   Var2   |     39008      |
|   Var3   |     39009      |
|   Var4   |     38717      |
|   Var5   |     38850      |
|   Var6   |      4417      |
|   Var7   |      4421      |
|   Var8   |     40000      |
|   Var9   |     39427      |
|  Var10   |     38850      |
|  Var11   |     39009      |
|  Var12   |     39543      |
|  Var13   |      4421      |
|  Var14   |     39009      |
|  Var15   |     40000      |
|  Var16   |     38850      |
|  Var17   |     38717      |
|  Var18   |     38717      |
|  Var19   |     38717      |
|  Var20   |     40000      |
|  Var21   |      4417      |
|  Var22   |      3998      |
|  Var23   |     38850      |
|  Var24   |      5779      |
|  Var25   |      3998      |
|  Var26   |     38850      |
|  Var27   |     38850      |
|  Var28   |      3999      |
|  Var29   |     39427      |
|  Var30   |     39427      |
|  Var31   |     40000      |
|  Var32   |     40000      |
|  Var33   |     39341      |
|  Var34   |     39008      |
|  Var35   |      3998      |
|  Var36   |     39008      |
|  Var37   |     38717      |
|  Var38   |      3998      |
|  Var39   |     40000      |
|  Var40   |     39008      |
|  Var41   |     39427      |
|  Var42   |     40000      |
|  Var43   |     39008      |
|  Var44   |      3998      |
|  Var45   |     39724      |
|  Var46   |     39008      |
|  Var47   |     39427      |
|  Var48   |     40000      |
|  Var49   |     39008      |
|  Var50   |     39427      |
|  Var51   |     37021      |
|  Var52   |     40000      |
|  Var53   |     39427      |
|  Var54   |     39008      |
|  Var55   |     40000      |
|  Var56   |     39481      |
|  Var57   |       0        |
|  Var58   |     39427      |
|  Var59   |     39338      |
|  Var60   |     38850      |
|  Var61   |     39341      |
```

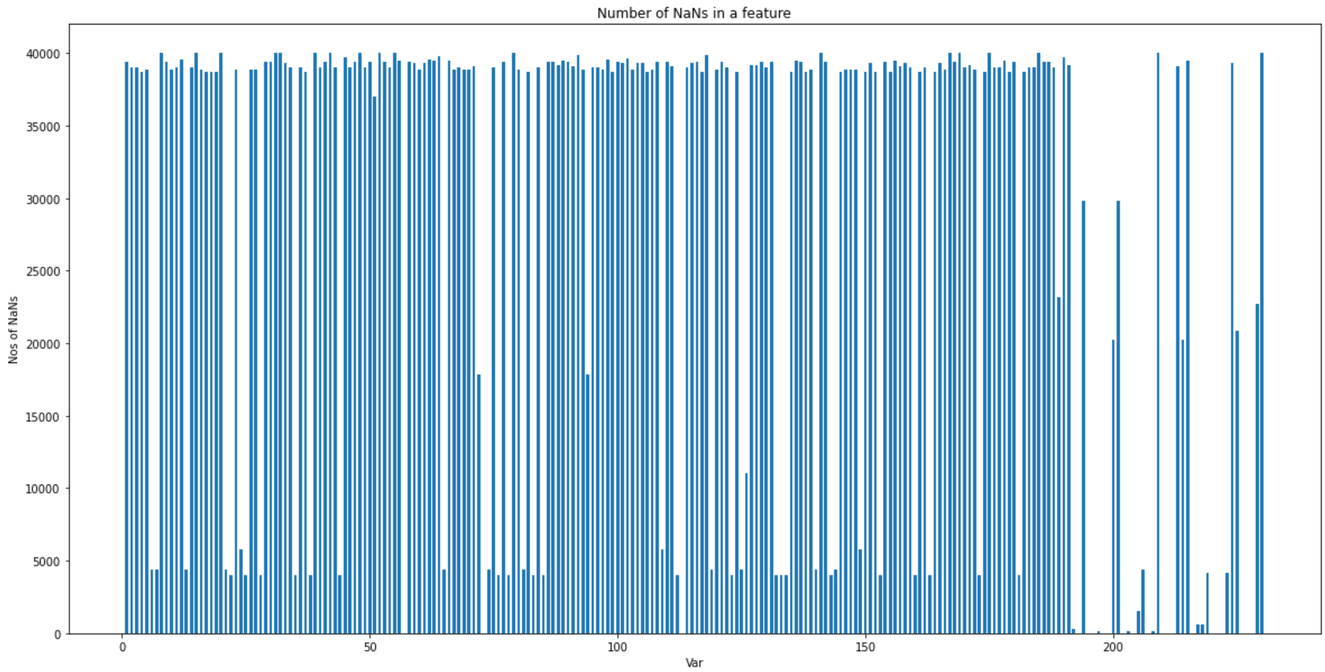
Var62	39543
Var63	39435
Var64	39813
Var65	4421
Var66	39435
Var67	38850
Var68	39008
Var69	38850
Var70	38850
Var71	39117
Var72	17807
Var73	0
Var74	4421
Var75	39008
Var76	3998
Var77	39427
Var78	3998
Var79	40000
Var80	38850
Var81	4417
Var82	38717
Var83	3998
Var84	39009
Var85	3998
Var86	39427
Var87	39427
Var88	39119
Var89	39481
Var90	39427
Var91	39117
Var92	39859
Var93	38850
Var94	17807
Var95	39008
Var96	39008
Var97	38850
Var98	39543
Var99	38717
Var100	39427
Var101	39296
Var102	39634
Var103	38850
Var104	39338
Var105	39338
Var106	38717
Var107	38850
Var108	39427
Var109	5779
Var110	39427
Var111	39117
Var112	3998
Var113	0
Var114	39008
Var115	39338
Var116	39427
Var117	38717
Var118	39859
Var119	4417
Var120	38850
Var121	39427
Var122	39008
Var123	3998
Var124	38717
Var125	4421
Var126	11059
Var127	39119
Var128	39119
Var129	39427
Var130	39009
Var131	39427
Var132	3998
Var133	3998
Var134	3998
Var135	38717
Var136	39435
Var137	39427
Var138	38717

Var139	38850
Var140	4421
Var141	40000
Var142	39427
Var143	3998
Var144	4417
Var145	38717
Var146	38850
Var147	38850
Var148	38850
Var149	5779
Var150	38717
Var151	39341
Var152	38717
Var153	3998
Var154	39427
Var155	38717
Var156	39435
Var157	39117
Var158	39296
Var159	39008
Var160	3998
Var161	38717
Var162	39008
Var163	3998
Var164	38717
Var165	39296
Var166	38850
Var167	40000
Var168	39427
Var169	40000
Var170	39008
Var171	39119
Var172	38850
Var173	3998
Var174	38717
Var175	40000
Var176	39009
Var177	39008
Var178	39481
Var179	38717
Var180	39427
Var181	3998
Var182	38717
Var183	39008
Var184	39008
Var185	40000
Var186	39427
Var187	39427
Var188	39008
Var189	23169
Var190	39735
Var191	39119
Var192	295
Var193	0
Var194	29771
Var195	0
Var196	0
Var197	110
Var198	0
Var199	4
Var200	20245
Var201	29772
Var202	1
Var203	110
Var204	0
Var205	1538
Var206	4417
Var207	0
Var208	110
Var209	40000
Var210	0
Var211	0
Var212	0
Var213	39117
Var214	20245
Var215	39435

```
| Var216  |      0       |
| Var217  |     570      |
| Var218  |     570      |
| Var219  |     4167     |
| Var220  |      0       |
| Var221  |      0       |
| Var222  |      0       |
| Var223  |     4167     |
| Var224  |    39338     |
| Var225  |    20842     |
| Var226  |      0       |
| Var227  |      0       |
| Var228  |      0       |
| Var229  |    22671     |
| Var230  |    40000     |
+----------+---------------+
```

In [ ]:

```
fig = plt.figure(figsize = (20, 10))
plt.bar(range(1,X_train_appetency.shape[1]+1), height = nan_count_array, width = 0.6,data = nan_count_a
rray)
plt.xlabel('Var')
plt.ylabel('Nos of NaNs')
plt.title('Number of NaNs in a feature')
plt.show()
```



Observation:

- Most of the features have high count of NaNs (near to 50k)
- Few features have NaN count under 10k
- Only a handful of features (belonging to categorical) have low or none count of NaNs
- There are 18 columns with only NaN value present

Unique value counts for Categorical features

- Categorical features ranges from Var191 to Var230
- We'll see the the unique values that each feature holds. Based on the count, we'll decide which categorical encoding to choose.
- If the value count per feature is high, then choosing OHE will result in highly sparse and large vectors.

In [ ]:

```
X_train_appetency.iloc[:,190:].head()
```

Out[ ]:

	Var191	Var192	Var193	Var194	Var195	Var196	Var197	Var198	Var199	Var20
1660	NaN	9rAq9at_88	rEUOq2QD1qfkRr6qpua	NaN	taul	1K8T	lK27	fhk21Ss	Tg7jjBB	60P9wL
24237	NaN	Qu0TmBQZiT	RO12	NaN	LfvqpCtLOY	1K8T	iJ4u	eQgxutV	n1zVHpT8NN	NaN
1673	NaN	zcROj1KVEH	RO12	NaN	taul	1K8T	0Y9G	T7ckueW	y2LIM01bE1	NaN
30043	r__I	dRavyx7ejg	RO12	NaN	taul	1K8T	SzjZ	z7269e2	r83_sZi	NaN
11297	NaN	1KSTmBQxuI	2Knk1KF	SEuy	taul	1K8T	PGNs	eDd7wZ4	gIRBFJT8NN	1qt7Yyi

In [ ]:

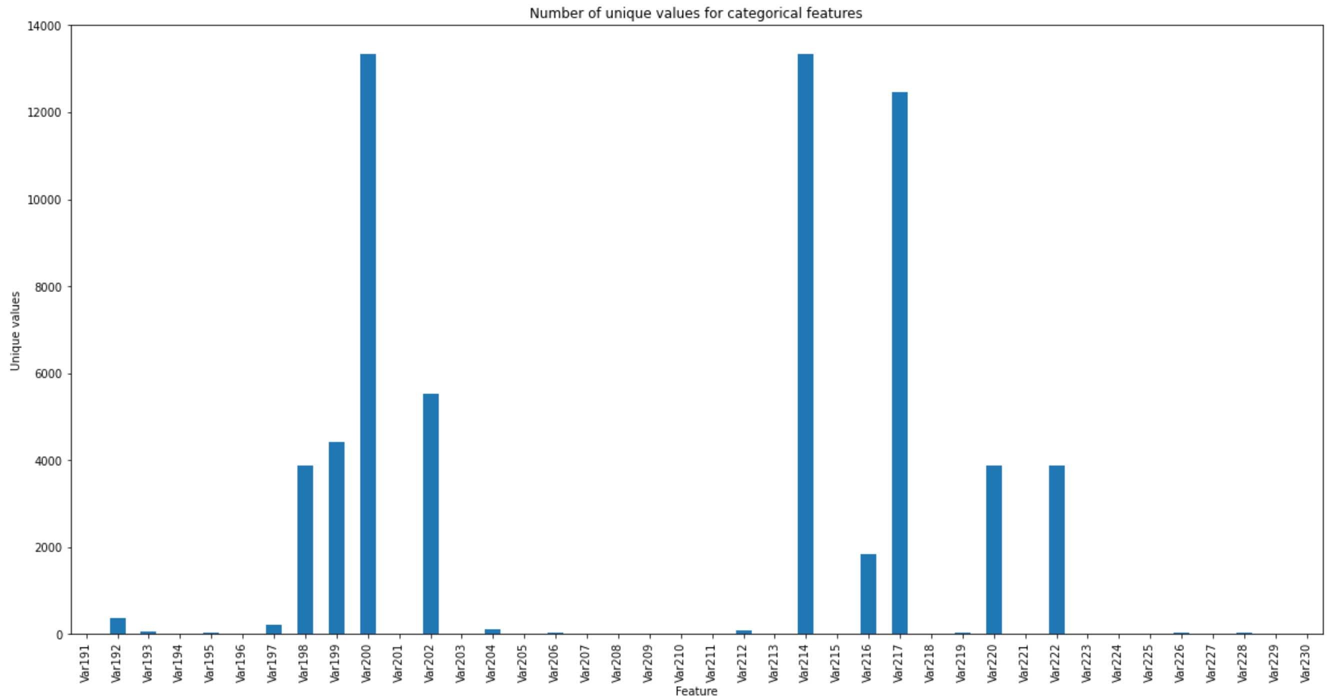
```
X_train_appetency.iloc[:,190:].nunique(axis = 0,dropna = False)
```

Out[ ]:

```
Var191        2
Var192      356
Var193       50
Var194        4
Var195       23
Var196        4
Var197      224
Var198     3879
Var199     4413
Var200    13351
Var201        3
Var202     5532
Var203        5
Var204      100
Var205        4
Var206       22
Var207       13
Var208        3
Var209        1
Var210        6
Var211        2
Var212       80
Var213        2
Var214    13351
Var215        2
Var216     1835
Var217    12474
Var218        3
Var219       23
Var220     3879
Var221        7
Var222     3879
Var223        5
Var224        2
Var225        4
Var226       23
Var227        7
Var228       30
Var229        5
Var230        1
dtype: int64
```

In [ ]:

```
#https://www.geeksforgeeks.org/how-to-count-distinct-values-of-a-pandas-dataframe-column/
fig = plt.figure(figsize = (20, 10))
X_train_appetency.iloc[:,190:].nunique(axis = 0,dropna = False).plot(kind = 'bar')
plt.title('Number of unique values for categorical features')
plt.xlabel('Feature')
plt.ylabel('Unique values')
plt.show()
```



Observation:

- 20 out of 40 feature have unique value count under 10.
- 9 features have unique value count which spans in range of 1000s

~~Although half of the categorical features have unique value count under 10, the other half has unique value counts in 1000s. It'll not be wise to apply OHE here as it'll create sparse vector having len in 1000s~~ The encoding method depends on the algorithm under consideration. For instance, LR works pretty well with high dimentional data. OHE could give a good score in this case. But the same might not be suitable for algorithms that are affected by curse of dimensionality like KNN. Also, it might not work well on tree based models.

Numerical range for class label

Since the number of numerical features are 190, we'll only look into range of few features.

We are only looking into features Var21 to Var25

In [ ]:

```
X_train_appetency.iloc[:,20:30].head()
```

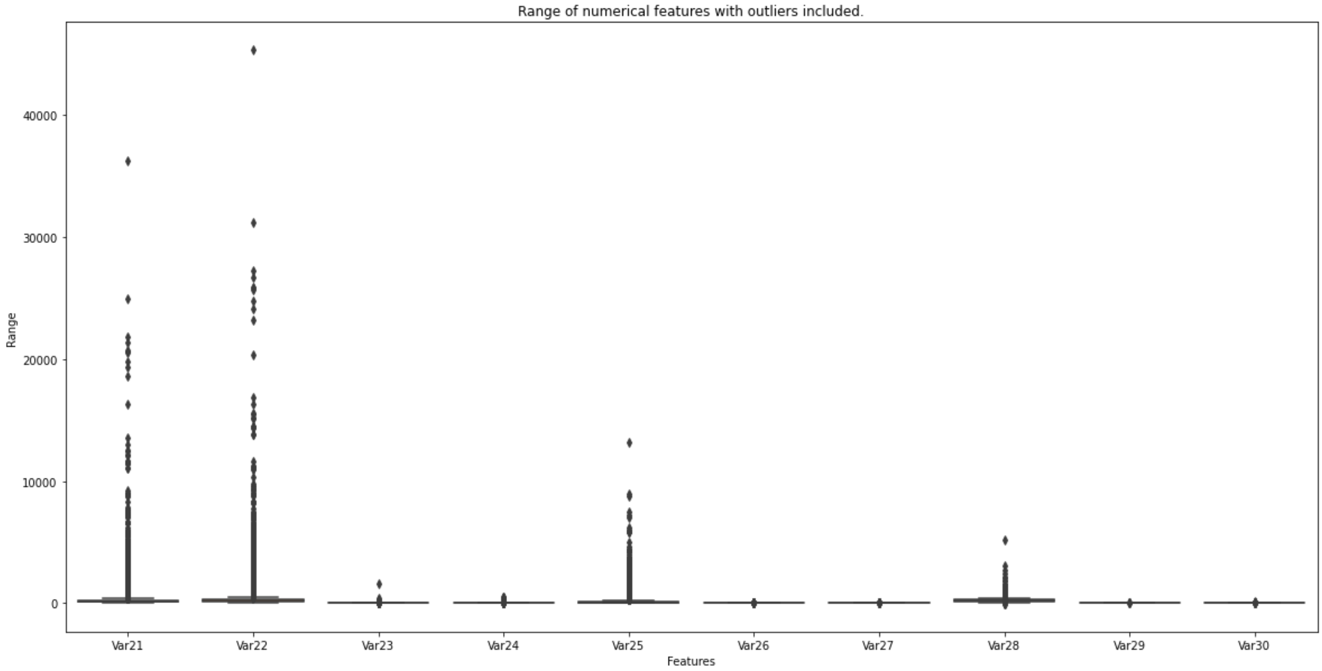
Out[ ]:

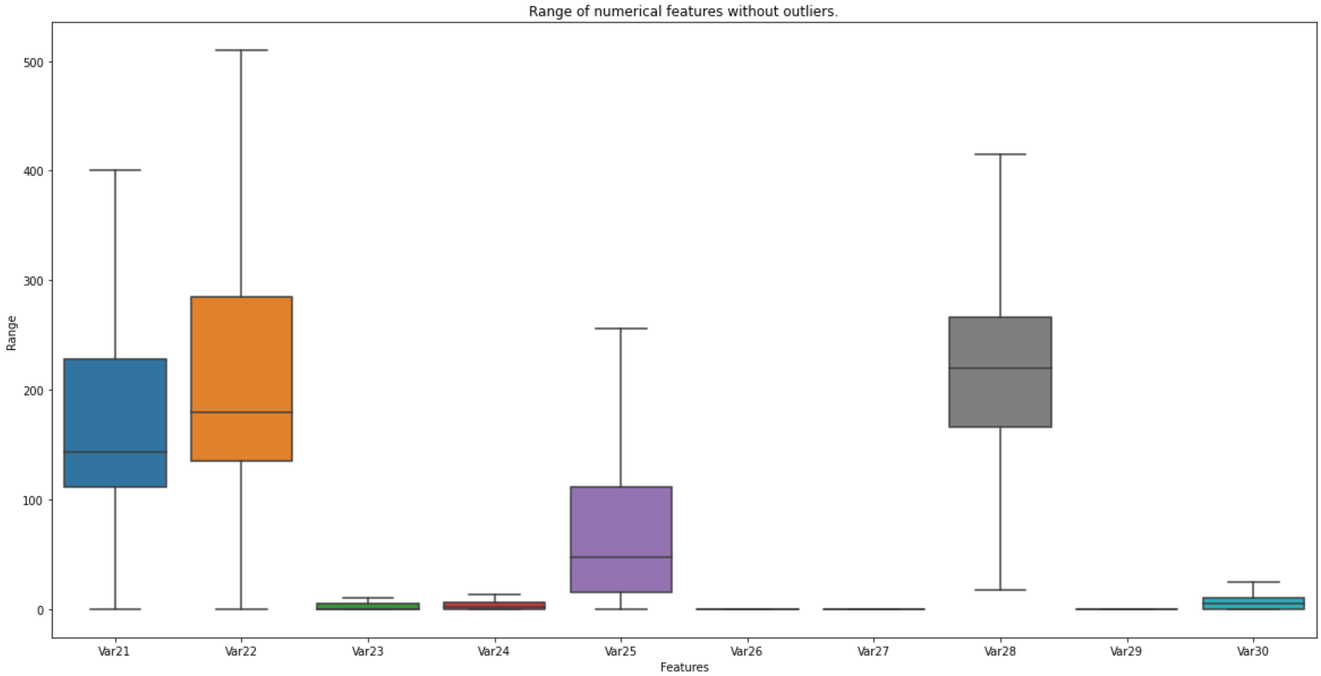
	Var21	Var22	Var23	Var24	Var25	Var26	Var27	Var28	Var29	Var30
1660	96.0	120.0	NaN	2.0	8.0	NaN	NaN	100.32	NaN	NaN
24237	12.0	15.0	NaN	0.0	24.0	NaN	NaN	238.72	NaN	NaN
1673	84.0	105.0	NaN	12.0	8.0	NaN	NaN	166.56	NaN	NaN
30043	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
11297	368.0	460.0	NaN	6.0	160.0	NaN	NaN	233.44	NaN	NaN

In [ ]:

```
# https://www.mikulskibartosz.name/how-to-remove-outliers-from-seaborn-boxplot-charts/
fig = plt.figure(figsize = (20, 10))
sns.boxplot(data = X_train_appetency.iloc[:,20:30])
plt.title('Range of numerical features with outliers included.')
plt.xlabel('Features')
```

```
plt.ylabel('Range')
plt.show()
fig = plt.figure(figsize = (20, 10))
sns.boxplot(data = X_train_appetency.iloc[:,20:30],showfliers = False)
plt.title('Range of numerical features without outliers.')
plt.xlabel('Features')
plt.ylabel('Range')
plt.show()
```





Observation:

- All features have different range.
- Var23 and Var24 have similar range.

Checking for pattern in missing values

In [ ]:

```
#https://towardsdatascience.com/missing-data-cfd9dbfd11b7
#https://towardsdatascience.com/all-about-missing-data-handling-b94b8b5d2184
#https://towardsdatascience.com/using-the-missingno-python-library-to-identify-and-visualise-missing-da
ta-prior-to-machine-learning-34c8c5b5f009
#https://github.com/ResidentMario/missingno
```

We'll check whether the NaNs value occur w.r.t a specific class or not

In [ ]:

```
data_with_labels = pd.concat([X_train_appetency,y_train_appetency],axis = 1)
```

In [ ]:

```
#https://stackoverflow.com/questions/53947196/groupby-class-and-count-missing-values-in-features
#https://stackoverflow.com/questions/39454542/divide-two-dataframes-with-python
# Percentage of NaNs w.r.t class
data_with_labels.isna().groupby(data_with_labels.Appetency).sum().div(data_with_labels.Appetency.value_
counts(),axis = 0) * 100
```

Out[ ]:

	Var1	Var2	Var3	Var4	Var5	Var6	Var7	Var8	Var9	Var10	
Appetency											
0	98.579719	97.541234	97.543779	96.808186	97.118713	10.995724	11.003360	100.0	98.579719	97.118713	9
1	97.893258	96.348315	96.348315	95.926966	97.471910	13.623596	13.764045	100.0	97.893258	97.471910	9

2 rows × 231 columns

Observation:

- NaN value doesn't occur for specific class.
- Percetage of NaN is uniform across classe.

Dropping columns with only NaNs value present

In [ ]:

```
temp_data = X_train_appetency.drop(columns = all_nan_columns)
```

In [ ]:

```
#temp_data = temp_data.drop(columns = not_nan_columns)
```

In [ ]:

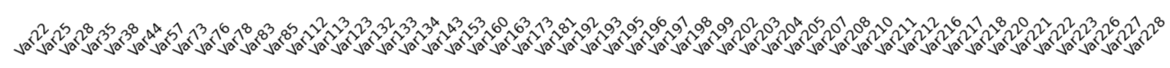
```
temp_data.shape
```

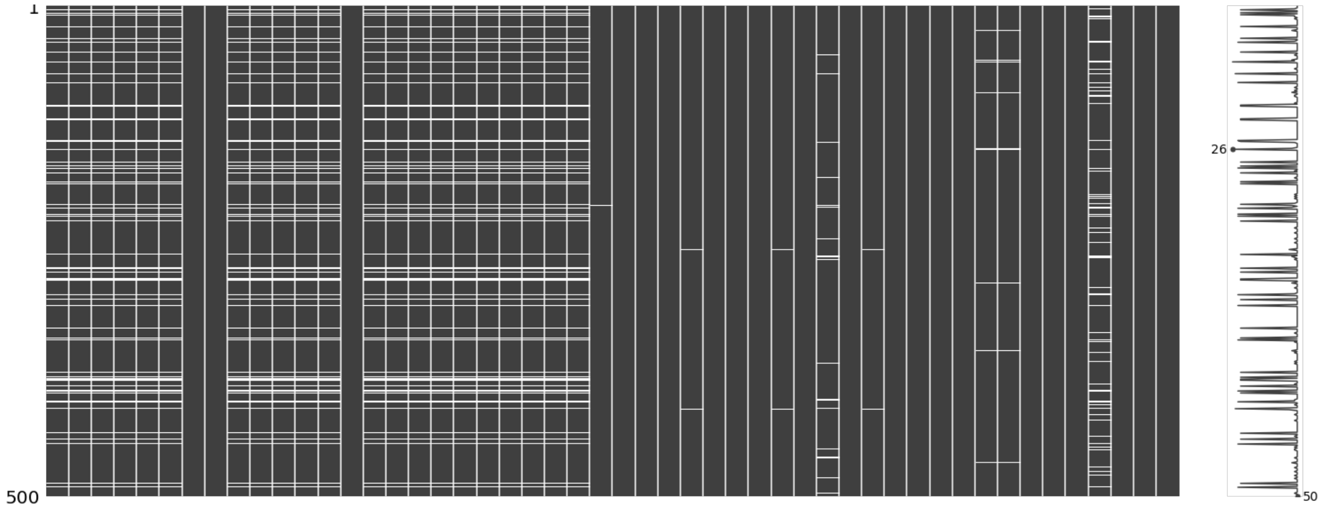
Out[ ]:

```
(40000, 212)
```

In [ ]:

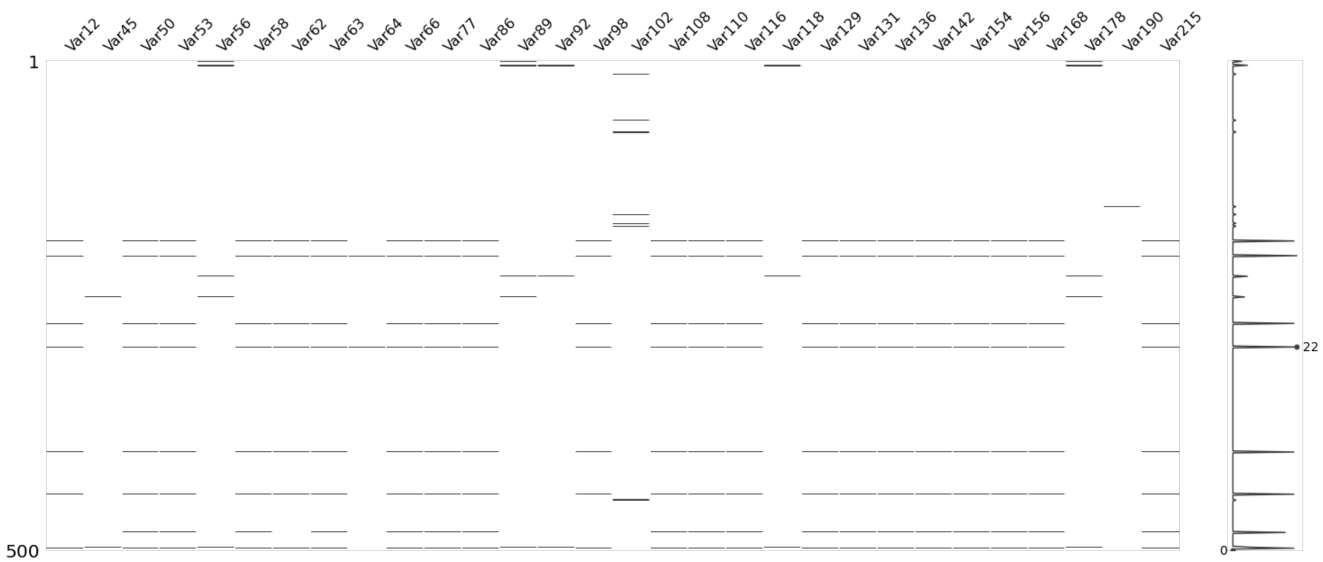
```
filter_data = msno.nullity_filter(temp_data, filter='top',n = 50)
msno.matrix(filter_data.sample(500))
plt.show()
```





In [ ]:

```
filter_data = msno.nullity_filter(temp_data, filter='bottom',n = 30)
msno.matrix(filter_data.sample(500))
plt.show()
```



The white lines in above figure represent missing data.

Observation:

- You can see there is a patten of missingness of values.

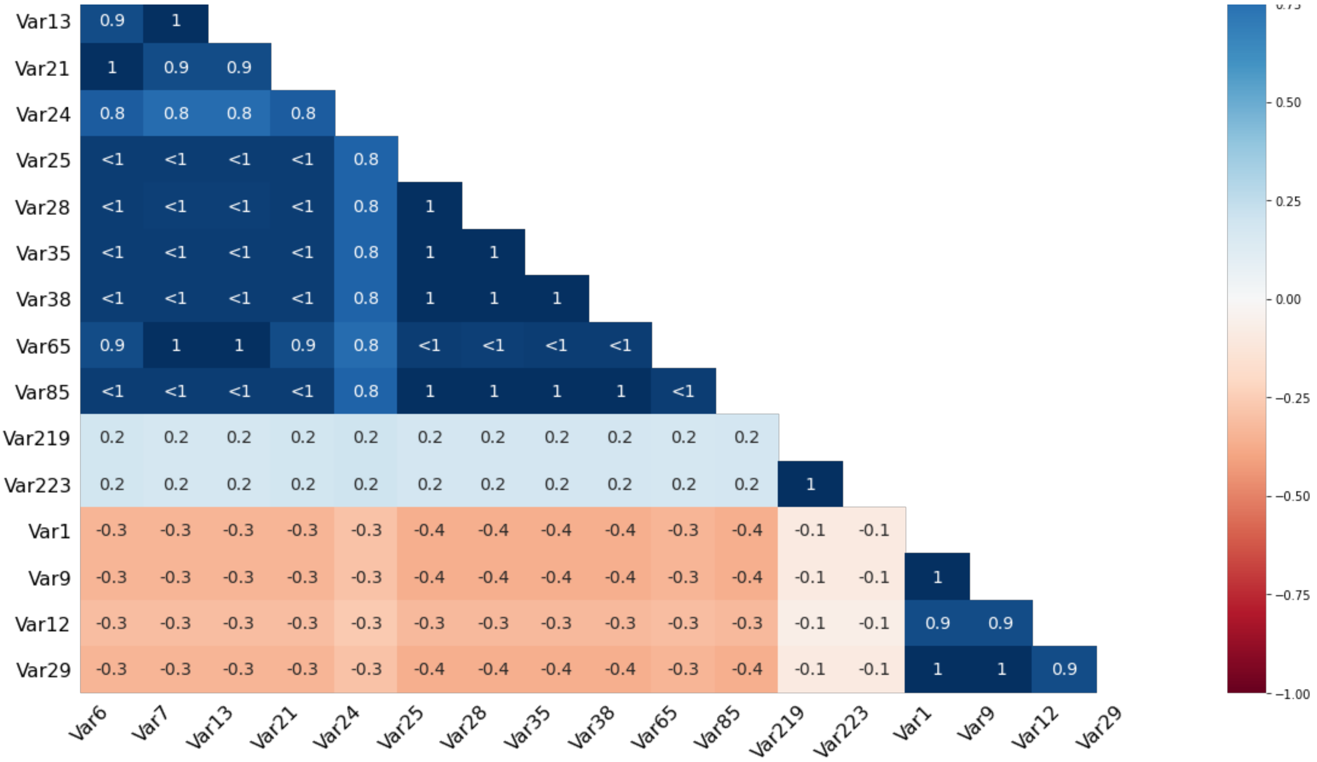
Checking for the correlation of missingness

In [ ]:

```
msno.heatmap(temp_data[['Var6','Var7','Var13','Var21','Var24','Var25','Var28','Var35','Var38','Var65','
Var85','Var219','Var223','Var1','Var9','Var12','Var29']])
```

Out[ ]:

<matplotlib.axes._subplots.AxesSubplot at 0x7f7baff2ad10>



We took a handful of features to check whether there is a correlation in missingness of values.

- Nullity correlation ranges from -1 (if one variable appears the other definitely does not) to 0 (variables appearing or not appearing have no effect on one another) to 1 (if one variable appears the other definitely also does).

Observation:

- Most of the features we checked have value 1 meaning there is a correlation in missingness

Conclusion:

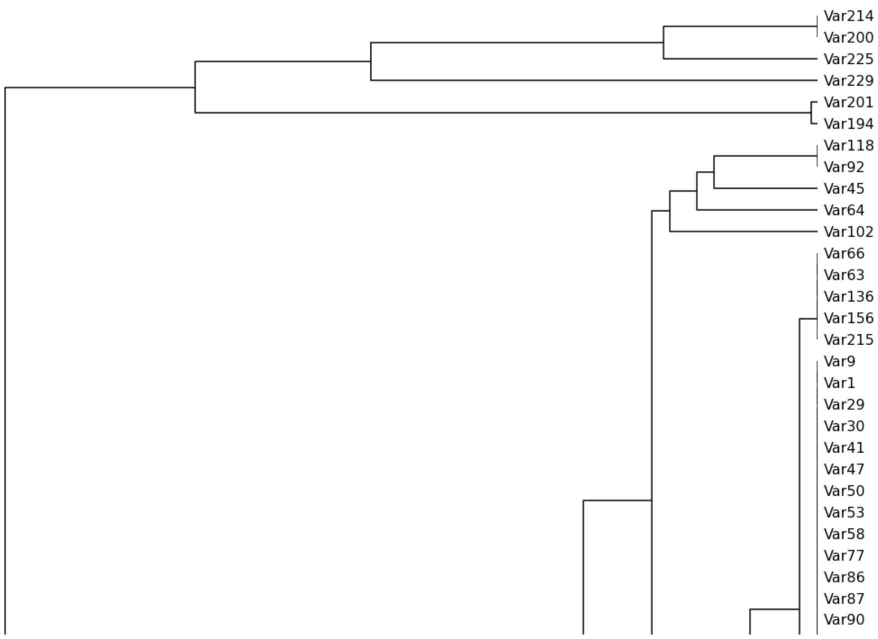
- As there is a correlation in missingness, we can rule out Missing Completely at random.

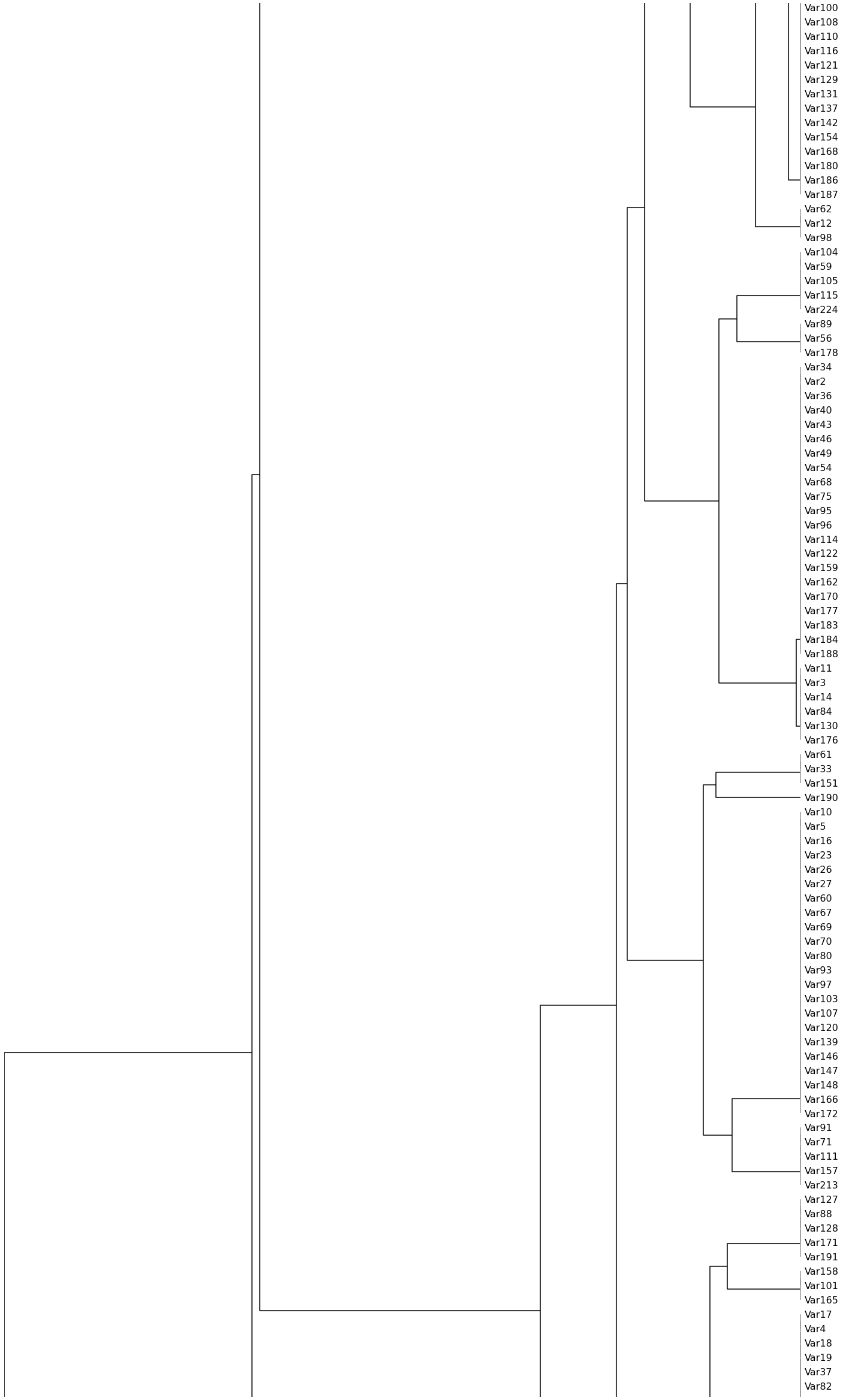
In [ ]:

```
msno.dendrogram(temp_data)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7bb0849610>
```




Var100
Var108
Var110
Var116
Var121
Var129
Var131
Var137
Var142
Var154
Var168
Var180
Var186
Var187
Var62
Var12
Var98
Var104
Var59
Var105
Var115
Var224
Var89
Var56
Var178
Var34
Var2
Var36
Var40
Var43
Var46
Var49
Var54
Var68
Var75
Var95
Var96
Var114
Var122
Var159
Var162
Var170
Var177
Var183
Var184
Var188
Var11
Var3
Var14
Var84
Var130
Var176
Var61
Var33
Var151
Var190
Var10
Var5
Var16
Var23
Var26
Var27
Var60
Var67
Var69
Var70
Var80
Var93
Var97
Var103
Var107
Var120
Var139
Var146
Var147
Var148
Var166
Var172
Var91
Var71
Var111
Var157
Var213
Var127
Var88
Var128
Var171
Var191
Var158
Var101
Var165
Var17
Var4
Var18
Var19
Var37
Var82

Observation:

- Variable value which are linked together at 0 fully predict one another's presence i.e one variable might always be empty when another is filled, or they might always both be filled or both empty, and so on.
- There are lot of variable which are linked at 0 distance.

- You can see from the matrix above that in a feature group there is a pattern of missingness

<font color= "red"> Query: how to conclude whether it is MAR (Missing at random) or MNAR(Missing not at random)?

https://www.youtube.com/watch?v=YpqUbirqFxQ

https://www.youtube.com/watch?v=ACN29i_fqkk

https://www.youtube.com/watch?v=asyJCVLV4LI

To check if the missing data depends on the observed data (MAR), we'll put sub sample of missing data columns against sample of categorical columns with no missing data and see if data is missing for specific categorical value.

In [ ]:

```
missing_data_cols = ['Var123','Var132','Var133','Var143','Var153','Var160','Var163','Var173','Var181']
#cat_cols = ['Var192','Var193','Var195','Var196','Var197','Var198','Var199','Var202','Var203','Var204']
```

In [ ]:

```
cat_not_nan_cols = not_nan_columns[3:]
```

In [ ]:

```
num_not_nan = not_nan_columns[:3]
```

In [ ]:

```
all_cols = missing_data_cols + list(cat_not_nan_cols)
```

In [ ]:

```
missing_data = X_train_appetency[all_cols]
```

In [ ]:

```
missing_data.head()
```

Out[ ]:

	Var123	Var132	Var133	Var143	Var153	Var160	Var163	Var173	Var181	Var193	Va
1660	42.0	0.0	130690.0	0.0	160024.0	28.0	33654.0	0.0	0.0	rEUOq2QD1qfkRr6qpua	taul
24237	0.0	0.0	0.0	0.0	21804.0	0.0	0.0	0.0	0.0	RO12	LfvqpC
1673	24.0	0.0	60000.0	0.0	128076.0	16.0	88896.0	0.0	0.0	RO12	taul
30043	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	RO12	taul
11297	66.0	0.0	1176270.0	0.0	1804564.0	82.0	213090.0	0.0	0.0	2Knk1KF	taul

In [ ]:

```
missing_data.isna().any(axis = 1)
```

Out[ ]:

```
1660     False
24237    False
1673     False
30043     True
11297    False
         ...
```

```
32360     True
8271     False
30452    False
26370    False
3279     False
Length: 40000, dtype: bool
```

In [ ]:

```
#https://stackoverflow.com/questions/14247586/how-to-select-rows-with-one-or-more-nulls-from-a-pandas-d
ataframe-without-listin
missing_data[missing_data.isna().any(axis = 1)][cat_not_nan_cols].nunique()
```

Out[ ]:

```
Var193       5
Var195       7
Var196       3
Var198    1126
Var204     100
Var207       6
Var210       5
Var211       2
Var212      15
Var216     150
Var220    1126
Var221       7
Var222    1126
Var226      23
Var227       7
Var228      10
dtype: int64
```

Checking for relation of missingness with numerical data

In [ ]:

```
num_not_nan
```

Out[ ]:

```
array(['Var57', 'Var73', 'Var113'], dtype=object)
```

In [ ]:

```
all_cols = missing_data_cols + list(num_not_nan)
```

In [ ]:

```
missing_data = X_train_appetency[all_cols]
```

In [ ]:

```
missing_data.head()
```

Out[ ]:

	Var123	Var132	Var133	Var143	Var153	Var160	Var163	Var173	Var181	Var57	Var73	Var113
1660	42.0	0.0	130690.0	0.0	160024.0	28.0	33654.0	0.0	0.0	2.917325	154	-1632648.0
24237	0.0	0.0	0.0	0.0	21804.0	0.0	0.0	0.0	0.0	2.773980	12	-1091284.0
1673	24.0	0.0	60000.0	0.0	128076.0	16.0	88896.0	0.0	0.0	2.089297	38	-1915512.0
30043	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.831233	8	-1010148.0
11297	66.0	0.0	1176270.0	0.0	1804564.0	82.0	213090.0	0.0	0.0	1.055330	166	-51767.2

Dropping all nan

In [ ]:

```
non_missing_data = missing_data.dropna()
```

Checking min and max

In [ ]:

```
non_missing_data.min()
```

Out[ ]:

```
Var123    0.000000e+00
Var132    0.000000e+00
Var133    0.000000e+00
Var143    0.000000e+00
Var153    0.000000e+00
Var160    0.000000e+00
Var163    0.000000e+00
Var173    0.000000e+00
Var181    0.000000e+00
Var57     2.136296e-04
Var73     1.200000e+01
Var113   -9.803600e+06
dtype: float64
```

In [ ]:

```
non_missing_data.max()
```

Out[ ]:

```
Var123       13086.0
Var132         160.0
Var133    15009900.0
Var143          18.0
Var153    13907800.0
Var160        4862.0
Var163    14515200.0
Var173           6.0
Var181          49.0
Var57            7.0
Var73          264.0
Var113     9932480.0
dtype: float64
```

Only keeping nan data and then checking min and max of numerical var

In [ ]:

```
missing = missing_data[missing_data.isna().any(axis =1)].iloc[:,-3:]
```

In [ ]:

```
missing_data[missing_data.isna().any(axis =1)].min()
```

Out[ ]:

```
Var123            NaN
Var132            NaN
Var133            NaN
Var143            NaN
Var153            NaN
Var160            NaN
```

```
Var163              NaN
Var173              NaN
Var181              NaN
Var57      2.136296e-04
Var73      4.000000e+00
Var113    -9.684120e+06
dtype: float64
```

In [ ]:

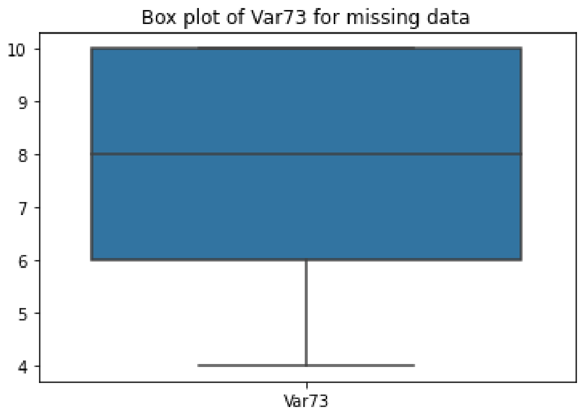
```python
missing_data[missing_data.isna().any(axis =1)].max()
```

Out[ ]:

```
Var123           NaN
Var132           NaN
Var133           NaN
Var143           NaN
Var153           NaN
Var160           NaN
Var163           NaN
Var173           NaN
Var181           NaN
Var57            7.0
Var73           10.0
Var113     6239680.0
dtype: float64
```

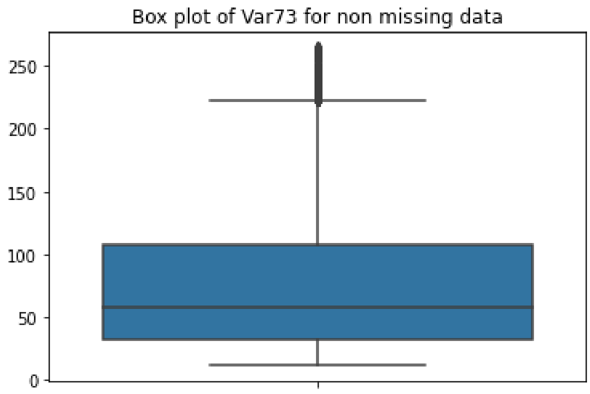
In [ ]:

```python
sns.boxplot(data = missing[['Var73']])
plt.title('Box plot of Var73 for missing data')
plt.show()
```



In [ ]:

```python
sns.boxplot(data = non_missing_data[['Var73']])
plt.title('Box plot of Var73 for non missing data')
plt.show()
```



Var73

Observation:

- For var73, if you look closely for non missing min and max, it is 12 and 264 resp. However max for missing data is 10.

We can say that for the data missing the value of Var73 end at 10 but for data present value of Var73 starts at 12. This may be one of many other cases present in dataset.

Since there is pattern in missingness and a missingness depends on observed data and we can assume that this is Missing at Random (MAR).

Now that we have concluded that data is Missing at Random (MAR), we can either remove the NaN data or we can use imputation.

For removing data, we have:

- Listwise deletion : Removes all data from an observation that has one or more missing values. Produces bias
- Pairwise deletion : Used in MCAR.
- Dropping variable : Dropping variables with having missing values %greater than 60%

We'll be dropping variables followed by imputation.

Reference : https://towardsdatascience.com/how-to-handle-missing-data-8646b18db0d4

Handling NaNs

As we can see from the graph above, most of features have NaN values reaching close to 40k out of 40k datapoints. In order to handle that, we'll be removing features in which NaN value exceeds the threshold. We'll check for 50,60, 70, 80 percent for threshold value.

In [ ]:

```
nan_count_array = np.asarray(nan_count_array)
```

In [ ]:

```
print('Number of features which have NaN count less than 50 perc of original data: ',(nan_count_array <
.5*X_train_appetency.shape[0]).sum())
```

```
Number of features which have NaN count less than 50 perc of original data:  69
```

In [ ]:

```
print('Number of features which have NaN count less than 60 perc of original data: ',(nan_count_array <
.6*X_train_appetency.shape[0]).sum())
```

```
Number of features which have NaN count less than 60 perc of original data:  74
```

In [ ]:

```
print('Number of features which have NaN count less than 70 perc of original data: ',(nan_count_array <
.7*X_train_appetency.shape[0]).sum())
```

```
Number of features which have NaN count less than 70 perc of original data:  74
```

In [ ]:

```
print('Number of features which have NaN count less than 80 perc of original data: ',(nan_count_array <
.8*X_train_appetency.shape[0]).sum())
```

```
Number of features which have NaN count less than 80 perc of original data:  76
```

Observation:

- **When threshold is set at 50 perc, only 69 features have NaN count less than 50% of total data.**
- **For both 60 and 70 value of threshold, number of features remains same at 74.**
- **When threshold is set at 80%, number of features that satify the condition are 76. An increase of two feature from last observation.**

We'll continue with 60% threshold and remove features which have NaN count more than 60%

In [ ]:

```
features = np.argwhere(nan_count_array < .6*X_train_appetency.shape[0])
```

In [ ]:

```
features = features.flatten()
```

In [ ]:

```
features
```

Out[ ]:

```
array([  5,   6,  12,  20,  21,  23,  24,  27,  34,  37,  43,  56,  64,
        71,  72,  73,  75,  77,  80,  82,  84,  93, 108, 111, 112, 118,
       122, 124, 125, 131, 132, 133, 139, 142, 143, 148, 152, 159, 162,
       172, 180, 188, 191, 192, 194, 195, 196, 197, 198, 199, 201, 202,
       203, 204, 205, 206, 207, 209, 210, 211, 213, 215, 216, 217, 218,
       219, 220, 221, 222, 224, 225, 226, 227, 228])
```

In [ ]:

```
data_new = X_train_appetency.iloc[:,features]
data_new_test = X_test_appetency.iloc[:, features]
```

In [ ]:

```
X_test_appetency = X_test_appetency.iloc[:, features]
```

In [ ]:

```
data_new.head()
```

Out[ ]:

	Var6	Var7	Var13	Var21	Var22	Var24	Var25	Var28	Var35	Var38	Var44	Var57	Var65	Var72	Var73	V
1660	315.0	7.0	1900.0	96.0	120.0	2.0	8.0	100.32	0.0	22116.0	0.0	2.917325	9.0	NaN	154	0.
24237	0.0	0.0	0.0	12.0	15.0	0.0	24.0	238.72	0.0	0.0	0.0	2.773980	9.0	NaN	12	0.
1673	588.0	7.0	32.0	84.0	105.0	12.0	8.0	166.56	0.0	17556.0	0.0	2.089297	9.0	3.0	38	2
30043	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.831233	NaN	NaN	8	N
11297	3388.0	7.0	5084.0	368.0	460.0	6.0	160.0	233.44	0.0	209742.0	0.0	1.055330	9.0	3.0	166	6

In [ ]:

```
data_new_test.head()
```

Out[ ]:

	Var6	Var7	Var13	Var21	Var22	Var24	Var25	Var28	Var35	Var38	Var44	Var57	Var65	Var72	Var73	V
27046	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.613117	NaN	NaN	6	N
39155	623.0	0.0	0.0	172.0	215.0	4.0	24.0	220.08	0.0	2831868.0	0.0	1.439009	9.0	NaN	26	0.
31402	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.352703	NaN	NaN	4	N
14638	651.0	7.0	3292.0	120.0	150.0	0.0	0.0	220.08	0.0	0.0	0.0	2.941465	9.0	NaN	106	8
5274	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.307199	NaN	NaN	4	N

In [ ]:

```
#https://www.kaggle.com/questions-and-answers/181332
#http://shakedzy.xyz/dython/modules/nominal/#associations

# nominal.associations(data_new,figsize=(50,50), num_num_assoc= 'spearman',cmap = 'GnBu',mark_columns=T
rue);
```

Observation:

- **There are instances where a feature is highly correlated to other features. e.g : for Var21 has a correlation coef of 1 with Var22.**

<font color = 'red'> Query: Should we remove the highly correlated feature? i.e having corr > 0.8

This answer to this depends on factors like type of algorithm your are considering, interpretability of your results, etc.

Go through this thread once: https://datascience.stackexchange.com/questions/24452/in-supervised-learning-why-is-it-bad-to-have-correlated-features

Depending on the various experiment settings you create, treat the collinear features accordingly

We'll not be removing collinear features as having collinear features may or may not improve model performance but it will not degrade its performance. Also, they may be chance that new features based on these collinear features may add some new information to the model.

Feature Groups

Plotting means of the features

In [ ]:

```
data_new.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 40000 entries, 1660 to 3279
Data columns (total 74 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Var6    35583 non-null  float64
 1   Var7    35579 non-null  float64
 2   Var13   35579 non-null  float64
 3   Var21   35583 non-null  float64
 4   Var22   36002 non-null  float64
 5   Var24   34221 non-null  float64
 6   Var25   36002 non-null  float64
```

```
 7   Var28   36001 non-null  float64
 8   Var35   36002 non-null  float64
 9   Var38   36002 non-null  float64
 10  Var44   36002 non-null  float64
 11  Var57   40000 non-null  float64
 12  Var65   35579 non-null  float64
 13  Var72   22193 non-null  float64
 14  Var73   40000 non-null  int64
 15  Var74   35579 non-null  float64
 16  Var76   36002 non-null  float64
 17  Var78   36002 non-null  float64
 18  Var81   35583 non-null  float64
 19  Var83   36002 non-null  float64
 20  Var85   36002 non-null  float64
 21  Var94   22193 non-null  float64
 22  Var109  34221 non-null  float64
 23  Var112  36002 non-null  float64
 24  Var113  40000 non-null  float64
 25  Var119  35583 non-null  float64
 26  Var123  36002 non-null  float64
 27  Var125  35579 non-null  float64
 28  Var126  28941 non-null  float64
 29  Var132  36002 non-null  float64
 30  Var133  36002 non-null  float64
 31  Var134  36002 non-null  float64
 32  Var140  35579 non-null  float64
 33  Var143  36002 non-null  float64
 34  Var144  35583 non-null  float64
 35  Var149  34221 non-null  float64
 36  Var153  36002 non-null  float64
 37  Var160  36002 non-null  float64
 38  Var163  36002 non-null  float64
 39  Var173  36002 non-null  float64
 40  Var181  36002 non-null  float64
 41  Var189  16831 non-null  float64
 42  Var192  39705 non-null  object
 43  Var193  40000 non-null  object
 44  Var195  40000 non-null  object
 45  Var196  40000 non-null  object
 46  Var197  39890 non-null  object
 47  Var198  40000 non-null  object
 48  Var199  39996 non-null  object
 49  Var200  19755 non-null  object
 50  Var202  39999 non-null  object
 51  Var203  39890 non-null  object
 52  Var204  40000 non-null  object
 53  Var205  38462 non-null  object
 54  Var206  35583 non-null  object
 55  Var207  40000 non-null  object
 56  Var208  39890 non-null  object
 57  Var210  40000 non-null  object
 58  Var211  40000 non-null  object
 59  Var212  40000 non-null  object
 60  Var214  19755 non-null  object
 61  Var216  40000 non-null  object
 62  Var217  39430 non-null  object
 63  Var218  39430 non-null  object
 64  Var219  35833 non-null  object
 65  Var220  40000 non-null  object
 66  Var221  40000 non-null  object
 67  Var222  40000 non-null  object
 68  Var223  35833 non-null  object
 69  Var225  19158 non-null  object
 70  Var226  40000 non-null  object
 71  Var227  40000 non-null  object
 72  Var228  40000 non-null  object
 73  Var229  17329 non-null  object
dtypes: float64(41), int64(1), object(32)
memory usage: 22.9+ MB
```

In [ ]:

```
numerical_data = data_new.iloc[:,0:42]
numerical_data_test = data_new_test.iloc[:,:42]
```

In [ ]:

```
numerical_data.head()
```

Out[ ]:

	Var6	Var7	Var13	Var21	Var22	Var24	Var25	Var28	Var35	Var38	Var44	Var57	Var65	Var72	Var73	V
1660	315.0	7.0	1900.0	96.0	120.0	2.0	8.0	100.32	0.0	22116.0	0.0	2.917325	9.0	NaN	154	0.
24237	0.0	0.0	0.0	12.0	15.0	0.0	24.0	238.72	0.0	0.0	0.0	2.773980	9.0	NaN	12	0.
1673	588.0	7.0	32.0	84.0	105.0	12.0	8.0	166.56	0.0	17556.0	0.0	2.089297	9.0	3.0	38	2
30043	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.831233	NaN	NaN	8	N
11297	3388.0	7.0	5084.0	368.0	460.0	6.0	160.0	233.44	0.0	209742.0	0.0	1.055330	9.0	3.0	166	6

In [ ]:

```
numerical_data_test.head()
```

Out[ ]:

	Var6	Var7	Var13	Var21	Var22	Var24	Var25	Var28	Var35	Var38	Var44	Var57	Var65	Var72	Var73	V
27046	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.613117	NaN	NaN	6	N
39155	623.0	0.0	0.0	172.0	215.0	4.0	24.0	220.08	0.0	2831868.0	0.0	1.439009	9.0	NaN	26	0.
31402	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.352703	NaN	NaN	4	N
14638	651.0	7.0	3292.0	120.0	150.0	0.0	0.0	220.08	0.0	0.0	0.0	2.941465	9.0	NaN	106	8
5274	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.307199	NaN	NaN	4	N

In [ ]:

```
numerical_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 40000 entries, 1660 to 3279
Data columns (total 42 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Var6    35583 non-null  float64
 1   Var7    35579 non-null  float64
 2   Var13   35579 non-null  float64
 3   Var21   35583 non-null  float64
 4   Var22   36002 non-null  float64
 5   Var24   34221 non-null  float64
 6   Var25   36002 non-null  float64
 7   Var28   36001 non-null  float64
 8   Var35   36002 non-null  float64
 9   Var38   36002 non-null  float64
 10  Var44   36002 non-null  float64
 11  Var57   40000 non-null  float64
 12  Var65   35579 non-null  float64
 13  Var72   22193 non-null  float64
 14  Var73   40000 non-null  int64
 15  Var74   35579 non-null  float64
 16  Var76   36002 non-null  float64
 17  Var78   36002 non-null  float64
 18  Var81   35583 non-null  float64
 19  Var83   36002 non-null  float64
```

```
 20  Var85    36002 non-null  float64
 21  Var94    22193 non-null  float64
 22  Var109   34221 non-null  float64
 23  Var112   36002 non-null  float64
 24  Var113   40000 non-null  float64
 25  Var119   35583 non-null  float64
 26  Var123   36002 non-null  float64
 27  Var125   35579 non-null  float64
 28  Var126   28941 non-null  float64
 29  Var132   36002 non-null  float64
 30  Var133   36002 non-null  float64
 31  Var134   36002 non-null  float64
 32  Var140   35579 non-null  float64
 33  Var143   36002 non-null  float64
 34  Var144   35583 non-null  float64
 35  Var149   34221 non-null  float64
 36  Var153   36002 non-null  float64
 37  Var160   36002 non-null  float64
 38  Var163   36002 non-null  float64
 39  Var173   36002 non-null  float64
 40  Var181   36002 non-null  float64
 41  Var189   16831 non-null  float64
dtypes: float64(41), int64(1)
memory usage: 13.1 MB
```

In [ ]:

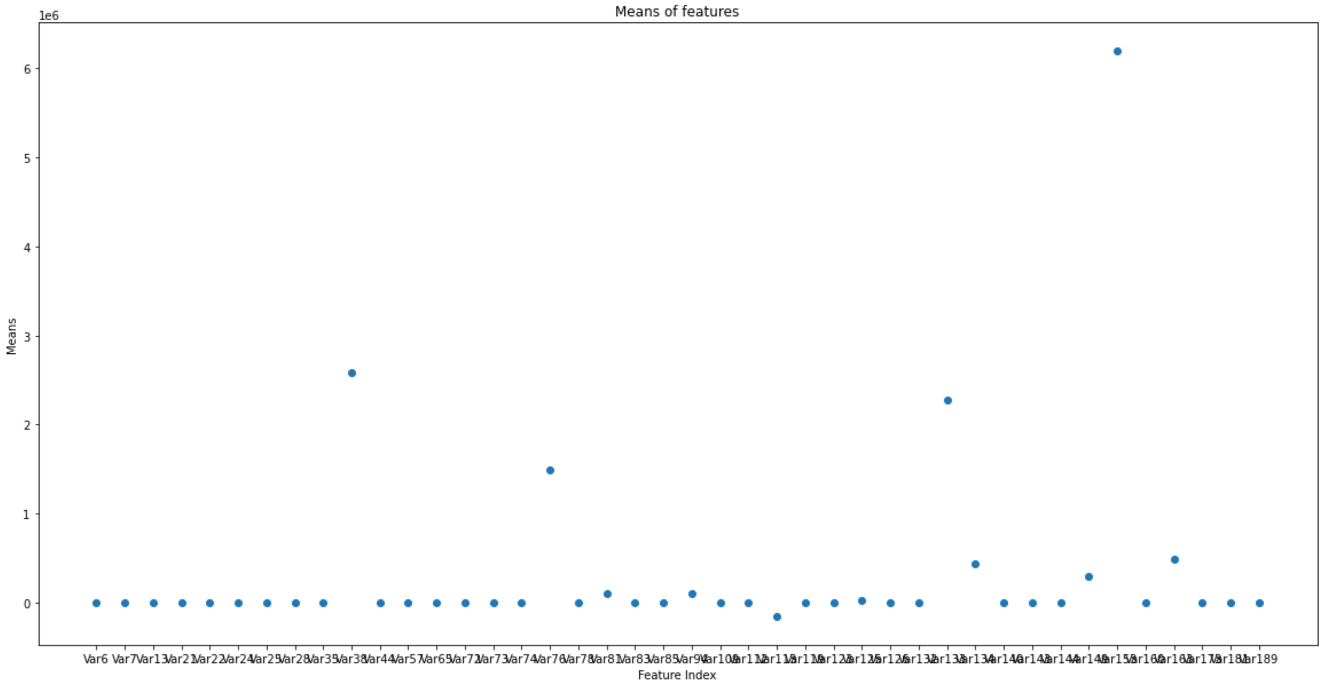
```
means = numerical_data.mean()
```

In [ ]:

```
means_test = numerical_data_test.mean()
```

In [ ]:

```
plt.figure(figsize = (20, 10))
plt.scatter(numerical_data.columns,means)
plt.title('Means of features')
plt.xlabel('Feature Index')
plt.ylabel('Means')
plt.show()
```



Observation:

- Most of means on scale are close to 0.
- Only 4 features have mean > 1 million

Let's try again by removing means > 1000000

In [ ]:

```
filter_means = means[means < 1000000]
```

In [ ]:

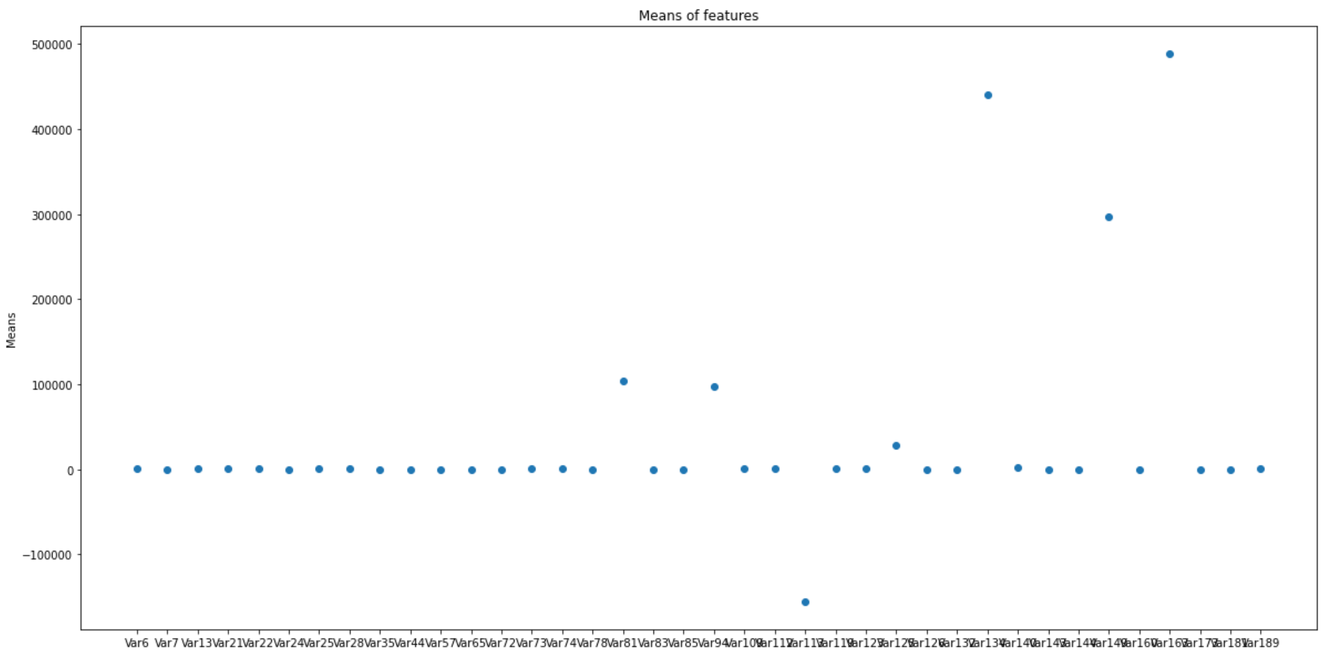
```
filter_means.shape
```

Out[ ]:

```
(38,)
```

In [ ]:

```
plt.figure(figsize = (20, 10))
plt.scatter(filter_means.index, filter_means)
plt.title('Means of features')
plt.ylabel('Means')
plt.show()
```



Observation:

- Out of 42 numerical features, 38 are under mean of 1 million
- Most of the means are concentrated in region < 1 mil and close to 0

Let's plot region under 10k

In [ ]:

```
filter_means = means[(means < 10000) & (means > 0)]
```

In [ ]:

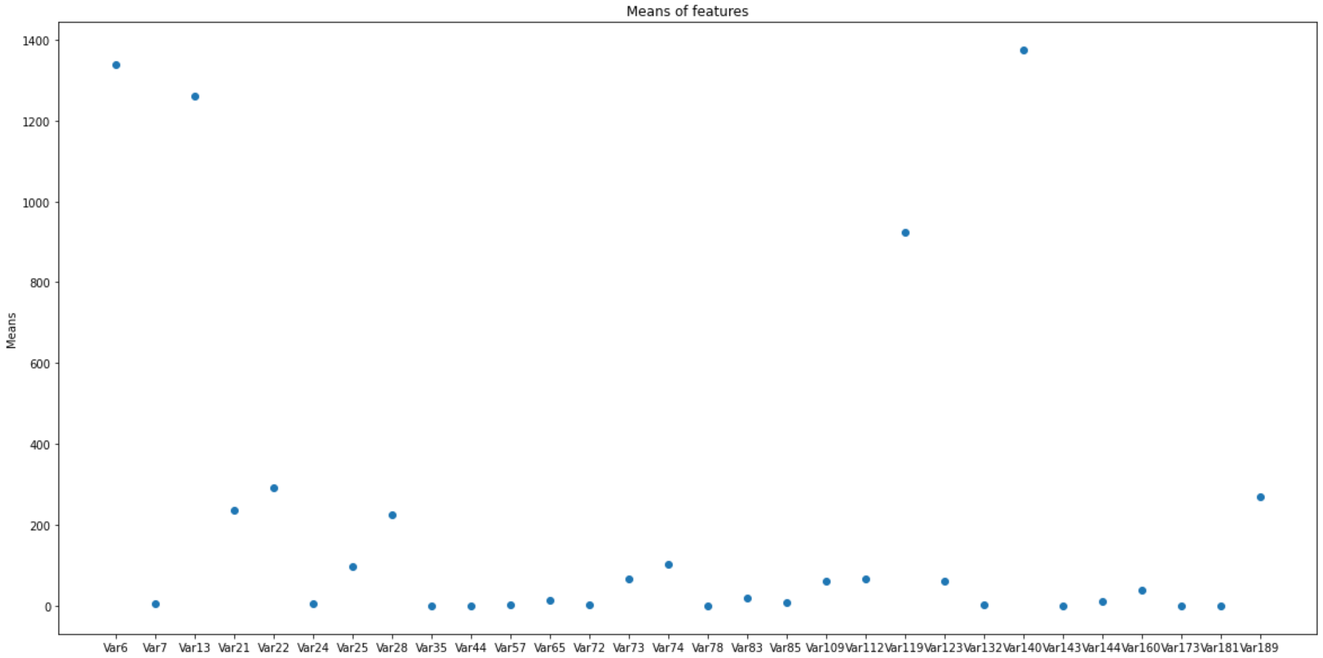
```
filter_means.shape
```

Out[ ]:

```
(30,)
```

In [ ]:

```
plt.figure(figsize = (20, 10))
plt.scatter(filter_means.index, filter_means)
plt.title('Means of features')
plt.ylabel('Means')
plt.show()
```



Observation:

- **There are 30 points which lie under 10k.**
- **Most of the points are concentrated under 400**

let's observation area under mean of 400

In [ ]:

```
filter_means = means[(means < 400) & (means > 0)]
```

In [ ]:

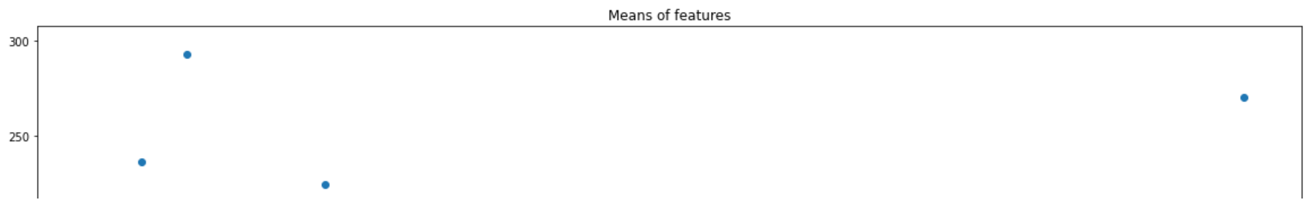
```
filter_means.shape
```

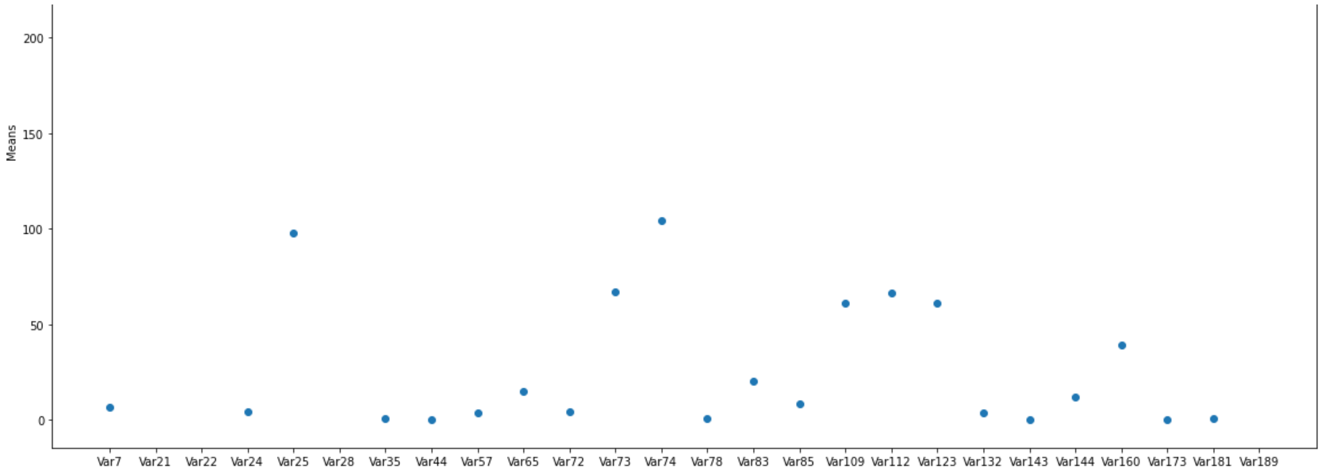
Out[ ]:

```
(26,)
```

In [ ]:

```
plt.figure(figsize = (20, 10))
plt.scatter(filter_means.index, filter_means)
plt.title('Means of features')
plt.ylabel('Means')
plt.show()
```





Observation:

- **Most of the means are concentrated under 50.**

In [ ]:

```
filter_means = means[(means < 50) & (means > 0)]
```

In [ ]:

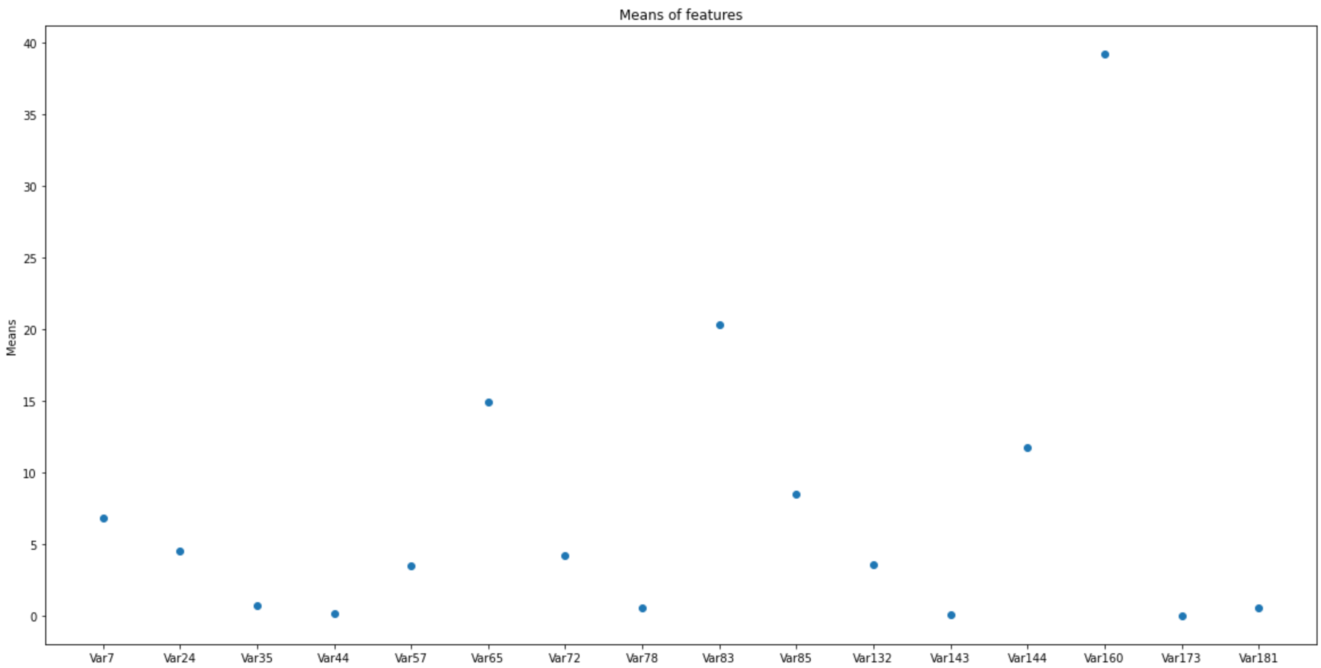
```
filter_means.shape
```

Out[ ]:

```
(16,)
```

In [ ]:

```
plt.figure(figsize = (20, 10))
plt.scatter(filter_means.index, filter_means)
plt.title('Means of features')
plt.ylabel('Means')
plt.show()
```



Observation:

- Out of 42 numerical features, 26 have mean under 400.
- Out of 42 numerical features, 16 features have mean under 50.
- 14 of the features have mean under 20.

<font color= 'red'>Query: How does feature groups help us ?

```
Insight could help you create new features.
```

<font color = 'red'>Query: How does means help in identifying feature groups ?

```
We can form a feature group for features having similar means and use that feature group to gene
rate new features. for e.g: a a new feature which is average value of features having mean under
20.
```

We'll be making 2 new feature groups i.e

1. Features having means under 200 and greater than 0
2. Features having means under 20 and greater than 0

In [ ]:

```
means_test
```

Out[ ]:

```
Var6       1.280190e+03
Var7       6.761202e+00
Var13      1.210703e+03
Var21      2.266976e+02
Var22      2.803243e+02
Var24      4.527547e+00
Var25      9.384804e+01
Var28      2.243610e+02
Var35      6.952942e-01
Var38      2.555882e+06
Var44      1.601958e-01
Var57      3.548064e+00
Var65      1.472608e+01
Var72      4.189055e+00
Var73      6.587980e+01
Var74      1.011020e+02
Var76      1.492063e+06
Var78      5.196351e-01
Var81      1.027725e+05
Var83      1.902047e+01
Var85      8.363110e+00
Var94      1.058181e+05
Var109     6.074348e+01
Var112     6.557615e+01
Var113    -1.425415e+05
Var119     8.828949e+02
Var123     5.699566e+01
Var125     2.717455e+04
Var126    -5.250035e-01
Var132     3.445100e+00
Var133     2.263096e+06
Var134     4.242102e+05
Var140     1.408799e+03
Var143     4.405384e-02
Var144     1.167934e+01
Var149     2.874525e+05
Var153     6.125481e+06
Var160     3.730382e+01
Var163     4.753434e+05
Var173     5.562354e-03
Var181     6.331071e-01
Var189     2.693973e+02
dtype: float64
```

In [ ]:

```
feature_group_200 = means[(means < 200) & (means > 0)]
```

In [ ]:

```
feature_group_200 = list(feature_group_200.index)
```

In [ ]:

```
feature_group_50 = means[(means < 50) & (means > 0)]
```

In [ ]:

```
feature_group_50 = list(feature_group_50.index)
```

In [ ]:

```
with open('feature_group_200.pickle', 'wb') as handle:
    pickle.dump(feature_group_200, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

In [ ]:

```
with open('feature_group_50.pickle', 'wb') as handle:
    pickle.dump(feature_group_50, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

Clustering of features

In [ ]:

```
#https://medium.com/analytics-vidhya/gowers-distance-899f9c4bd553
#https://towardsdatascience.com/clustering-datasets-having-both-numerical-and-categorical-variables-ed9
1cdca0677
```

In [ ]:

```
data_new.head()
```

Out[ ]:

	Var6	Var7	Var13	Var21	Var22	Var24	Var25	Var28	Var35	Var38	Var44	Var57	Var65	Var72	Var73	V.
1660	315.0	7.0	1900.0	96.0	120.0	2.0	8.0	100.32	0.0	22116.0	0.0	2.917325	9.0	NaN	154	0.
24237	0.0	0.0	0.0	12.0	15.0	0.0	24.0	238.72	0.0	0.0	0.0	2.773980	9.0	NaN	12	0.
1673	588.0	7.0	32.0	84.0	105.0	12.0	8.0	166.56	0.0	17556.0	0.0	2.089297	9.0	3.0	38	2'
30043	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.831233	NaN	NaN	8	N
11297	3388.0	7.0	5084.0	368.0	460.0	6.0	160.0	233.44	0.0	209742.0	0.0	1.055330	9.0	3.0	166	6(

In [ ]:

```
data_new_test.head()
```

Out[ ]:

	Var6	Var7	Var13	Var21	Var22	Var24	Var25	Var28	Var35	Var38	Var44	Var57	Var65	Var72	Var73	V
27046	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.613117	NaN	NaN	6	N
39155	623.0	0.0	0.0	172.0	215.0	4.0	24.0	220.08	0.0	2831868.0	0.0	1.439009	9.0	NaN	26	0.
31402	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.352703	NaN	NaN	4	N
14638	651.0	7.0	3292.0	120.0	150.0	0.0	0.0	220.08	0.0	0.0	0.0	2.941465	9.0	NaN	106	8
5274	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.307199	NaN	NaN	4	N

Before we start off with clustering, we need to deal with NaN data. For numerical data, we'll perform mean imputation and for categorical data, we'll consider NaN as separate category.

In [ ]:

```
data_new.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 40000 entries, 1660 to 3279
Data columns (total 74 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Var6    35583 non-null  float64
 1   Var7    35579 non-null  float64
 2   Var13   35579 non-null  float64
 3   Var21   35583 non-null  float64
 4   Var22   36002 non-null  float64
 5   Var24   34221 non-null  float64
 6   Var25   36002 non-null  float64
 7   Var28   36001 non-null  float64
 8   Var35   36002 non-null  float64
 9   Var38   36002 non-null  float64
 10  Var44   36002 non-null  float64
 11  Var57   40000 non-null  float64
 12  Var65   35579 non-null  float64
 13  Var72   22193 non-null  float64
 14  Var73   40000 non-null  int64
 15  Var74   35579 non-null  float64
 16  Var76   36002 non-null  float64
 17  Var78   36002 non-null  float64
 18  Var81   35583 non-null  float64
 19  Var83   36002 non-null  float64
 20  Var85   36002 non-null  float64
 21  Var94   22193 non-null  float64
 22  Var109  34221 non-null  float64
 23  Var112  36002 non-null  float64
 24  Var113  40000 non-null  float64
 25  Var119  35583 non-null  float64
 26  Var123  36002 non-null  float64
 27  Var125  35579 non-null  float64
 28  Var126  28941 non-null  float64
 29  Var132  36002 non-null  float64
 30  Var133  36002 non-null  float64
 31  Var134  36002 non-null  float64
 32  Var140  35579 non-null  float64
 33  Var143  36002 non-null  float64
 34  Var144  35583 non-null  float64
 35  Var149  34221 non-null  float64
 36  Var153  36002 non-null  float64
 37  Var160  36002 non-null  float64
 38  Var163  36002 non-null  float64
 39  Var173  36002 non-null  float64
 40  Var181  36002 non-null  float64
 41  Var189  16831 non-null  float64
 42  Var192  39705 non-null  object
 43  Var193  40000 non-null  object
 44  Var195  40000 non-null  object
 45  Var196  40000 non-null  object
 46  Var197  39890 non-null  object
```

```
 47  Var198  40000 non-null  object
 48  Var199  39996 non-null  object
 49  Var200  19755 non-null  object
 50  Var202  39999 non-null  object
 51  Var203  39890 non-null  object
 52  Var204  40000 non-null  object
 53  Var205  38462 non-null  object
 54  Var206  35583 non-null  object
 55  Var207  40000 non-null  object
 56  Var208  39890 non-null  object
 57  Var210  40000 non-null  object
 58  Var211  40000 non-null  object
 59  Var212  40000 non-null  object
 60  Var214  19755 non-null  object
 61  Var216  40000 non-null  object
 62  Var217  39430 non-null  object
 63  Var218  39430 non-null  object
 64  Var219  35833 non-null  object
 65  Var220  40000 non-null  object
 66  Var221  40000 non-null  object
 67  Var222  40000 non-null  object
 68  Var223  35833 non-null  object
 69  Var225  19158 non-null  object
 70  Var226  40000 non-null  object
 71  Var227  40000 non-null  object
 72  Var228  40000 non-null  object
 73  Var229  17329 non-null  object
dtypes: float64(41), int64(1), object(32)
memory usage: 22.9+ MB
```

In [ ]:

```
data_new.mean()
```

Out[ ]:

```
Var6       1.337989e+03
Var7       6.821552e+00
Var13      1.259421e+03
Var21      2.364717e+02
Var22      2.927225e+02
Var24      4.503024e+00
Var25      9.757080e+01
Var28      2.245443e+02
Var35      7.221821e-01
Var38      2.584906e+06
Var44      1.684906e-01
Var57      3.503373e+00
Var65      1.490455e+01
Var72      4.191051e+00
Var73      6.683140e+01
Var74      1.042962e+02
Var76      1.489677e+06
Var78      5.384701e-01
Var81      1.031619e+05
Var83      2.027401e+01
Var85      8.485473e+00
Var94      9.692336e+04
Var109     6.092493e+01
Var112     6.638209e+01
Var113    -1.559629e+05
Var119     9.244093e+02
Var123     6.098511e+01
Var125     2.806564e+04
Var126    -5.610034e-01
Var132     3.544470e+00
Var133     2.276188e+06
Var134     4.406187e+05
Var140     1.374385e+03
Var143     6.149658e-02
Var144     1.173974e+01
Var149     2.967865e+05
Var153     6.196071e+06
Var160     3.917732e+01
Var163     4.887582e+05
```

```
Var173    7.166269e-03
Var181    6.060497e-01
Var189    2.703276e+02
dtype: float64
```

In [ ]:

```python
data_impute = data_new.iloc[:,0:42].fillna(data_new.mean())
```

In [ ]:

```python
data_impute_test = data_new_test.iloc[:,0:42].fillna(data_new_test.mean())
```

In [ ]:

```python
data_new_imputed = pd.concat([data_impute, data_new.iloc[:,42:].fillna('Others')], axis =1)
```

In [ ]:

```python
data_new_imputed_test = pd.concat([data_impute_test, data_new_test.iloc[:,42:].fillna('Others')], axis
=1)
```

In [ ]:

```python
data_new_imputed.head()
```

Out[ ]:

	Var6	Var7	Var13	Var21	Var22	Var24	Var25	Var28	Var35	
1660	315.000000	7.000000	1900.000000	96.000000	120.000000	2.000000	8.000000	100.320000	0.000000	2.211
24237	0.000000	0.000000	0.000000	12.000000	15.000000	0.000000	24.000000	238.720000	0.000000	0.000
1673	588.000000	7.000000	32.000000	84.000000	105.000000	12.000000	8.000000	166.560000	0.000000	1.755
30043	1337.988731	6.821552	1259.420669	236.471686	292.722488	4.503024	97.570802	224.544274	0.722182	2.584
11297	3388.000000	7.000000	5084.000000	368.000000	460.000000	6.000000	160.000000	233.440000	0.000000	2.097

In [ ]:

```python
data_new_imputed_test.head()
```

Out[ ]:

	Var6	Var7	Var13	Var21	Var22	Var24	Var25	Var28	Var35	V
27046	1280.190369	6.761202	1210.703445	226.69757	280.324285	4.527547	93.848036	224.361049	0.695294	2.555882
39155	623.000000	0.000000	0.000000	172.00000	215.000000	4.000000	24.000000	220.080000	0.000000	2.831868
31402	1280.190369	6.761202	1210.703445	226.69757	280.324285	4.527547	93.848036	224.361049	0.695294	2.555882
14638	651.000000	7.000000	3292.000000	120.00000	150.000000	0.000000	0.000000	220.080000	0.000000	0.000000
5274	1280.190369	6.761202	1210.703445	226.69757	280.324285	4.527547	93.848036	224.361049	0.695294	2.555882

Since our data contain both categorical and numerical features, we'll first convert our Categorical Data to numerical using ordinal encoding.

In [ ]:

```
encoder = OrdinalEncoder(handle_unknown = 'use_encoded_value', unknown_value = -1)
```

In [ ]:

```
data_new_imputed.iloc[:, 42:].head()
```

Out[ ]:

	Var192	Var193	Var195	Var196	Var197	Var198	Var199	Var200	Var202	V
1660	9rAq9at_88	rEUOq2QD1qfkRr6qpua	taul	1K8T	IK27	fhk21Ss	Tg7jjBB	60P9wLk	ympL	9_
24237	Qu0TmBQZiT	RO12	LfvqpCtLOY	1K8T	iJ4u	eQgxutV	n1zVHpT8NN	Others	9JQA	9_
1673	zcROj1KVEH	RO12	taul	1K8T	0Y9G	T7ckueW	y2LIM01bE1	Others	uFNB	9_
30043	dRavyx7ejg	RO12	taul	1K8T	SzjZ	z7269e2	r83_sZi	Others	oe6C	9_
11297	1KSTmBQxul	2Knk1KF	taul	1K8T	PGNs	eDd7wZ4	gIRBFJT8NN	1qt7Yyi	SxPy	9_

In [ ]:

```
encoder.fit(data_new_imputed.iloc[:,42:])
```

Out[ ]:

```
OrdinalEncoder(handle_unknown='use_encoded_value', unknown_value=-1)
```

In [ ]:

```
ordinal_features = encoder.transform(data_new_imputed.iloc[:,42:])
```

In [ ]:

```
ordinal_features_test = encoder.transform(data_new_imputed_test.iloc[:,42:])
```

In [ ]:

```
ordinal_features.shape
```

Out[ ]:

```
(40000, 32)
```

In [ ]:

```
ordinal_features_test.shape
```

Out[ ]:

```
(10000, 32)
```

In [ ]:

```
numerical_features = data_new_imputed.iloc[:,0:42].values
numerical_features_test = data_new_imputed_test.iloc[:,0:42].values
```

In [ ]:

```
numerical_features.shape
```

```
numerical_features.shape
```

Out[ ]:

```
(40000, 42)
```

In [ ]:

```
numerical_features_test.shape
```

Out[ ]:

```
(10000, 42)
```

In [ ]:

```
final_features = np.hstack((numerical_features, ordinal_features))
```

In [ ]:

```
final_features.shape
```

Out[ ]:

```
(40000, 74)
```

In [ ]:

```
final_features_test = np.hstack((numerical_features_test, ordinal_features_test))
```

In [ ]:

```
final_features_test.shape
```

Out[ ]:

```
(10000, 74)
```

Clustering of points

Reference: https://towardsdatascience.com/how-to-create-new-features-using-clustering-4ae772387290

In [ ]:

```
train_labels = []
test_labels = []
for i in range(2,7):
    kmeans = KMeans(n_clusters=i, n_jobs = -1)
    kmeans.fit(final_features)
    train_labels.append(kmeans.labels_)
    test_labels.append(kmeans.predict(final_features_test))
```

In [ ]:

```
# embedded_features = TSNE(n_jobs = -1).fit_transform(final_features)
```

In [ ]:

```
# for i in range(5):
#     plt.figure(figsize = (20,20))
#     plt.scatter(embedded_features[:,0], embedded_features[:,1], c= labels[i])
#     plt.title('Clustering of Features. Number of cluster: {}'.format(i+2))
#     plt.show()
```

Observation:

- **The above plot shows the datapoints divided in 2,3,4,5 and 6 cluster.**

We will use this cluster label as new feature.

<font color = 'red'> Query: How does clustering help in feature group?

you can assign cluster numbers to similar features (groups) to create a new feature. Some more areas can also be explored.

Finding Duplicate features

In [ ]:

```
#https://towardsdatascience.com/the-fastml-guide-9ada1bb761cf
duplicate_features = get_duplicate_features(data_new)
```

In [ ]:

```
duplicate_features.head()
```

Out[ ]:

	Desc	feature1	feature2
0	Duplicate Index	Var198	Var220
1	Duplicate Index	Var198	Var222
2	Duplicate Index	Var220	Var222

From the Description, we can see that although the values of two features are different but they occur at same index. Let's print them and see.

In [ ]:

```
data_new[data_new.Var198 == 'NldASpP'][['Var198','Var220','Var222']]
```

Out[ ]:

	Var198	Var220	Var222
21672	NldASpP	JFM1BiF	NKv4yOc
28761	NldASpP	JFM1BiF	NKv4yOc
2322	NldASpP	JFM1BiF	NKv4yOc
36377	NldASpP	JFM1BiF	NKv4yOc
17160	NldASpP	JFM1BiF	NKv4yOc
...	...	...	...
16980	NldASpP	JFM1BiF	NKv4yOc
40437	NldASpP	JFM1BiF	NKv4yOc
15538	NldASpP	JFM1BiF	NKv4yOc
46360	NldASpP	JFM1BiF	NKv4yOc
156	NldASpP	JFM1BiF	NKv4yOc

62 rows × 3 columns

In [ ]:

```
data_new[data_new.Var198 == 'ka_ns41'][['Var198','Var220','Var222']]
```

Out[ ]:

	Var198	Var220	Var222
22975	ka_ns41	1YVfGrO	fXVEsaq
45573	ka_ns41	1YVfGrO	fXVEsaq
38111	ka_ns41	1YVfGrO	fXVEsaq
21125	ka_ns41	1YVfGrO	fXVEsaq
35984	ka_ns41	1YVfGrO	fXVEsaq
...	...	...	...
45798	ka_ns41	1YVfGrO	fXVEsaq
24143	ka_ns41	1YVfGrO	fXVEsaq
31825	ka_ns41	1YVfGrO	fXVEsaq
16834	ka_ns41	1YVfGrO	fXVEsaq
16947	ka_ns41	1YVfGrO	fXVEsaq

95 rows × 3 columns

Observation:

- Although we didn't find any duplicate features but there are 3 features for which value are different but they have same mapping.
- For ex: For column Var198, value 'ka_ns41' always occur with '1YVfGrO' (Var220) and 'fXVEsaq' (Var222)

<font color = 'red'> Query: Do we remove features with values having same mapping. If so, why?

The duplicate columns could be dropped Because they are the same things

Dropping Var220 and Var222

In [ ]:

```
data_new = data_new.drop(['Var220','Var222'], axis = 1)
```

In [ ]:

```
data_new.shape
```

Out[ ]:

```
(40000, 72)
```

In [ ]:

```
X_test_appetency =  X_test_appetency.drop(['Var220','Var222'], axis = 1)
```

In [ ]:

```
X_test_appetency.shape
```

Out[ ]:

```
(10000, 72)
```

2 columns have been dropped from dataset. We're left with 72 features now instead of 74

Saving data in pickle file

In [ ]:

```
with open('X_train_appetency.pickle', 'wb') as handle:
    pickle.dump(data_new, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

In [ ]:

```
with open('y_train_appetency.pickle', 'wb') as handle:
    pickle.dump(y_train_appetency, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

In [ ]:

```
with open('X_test_appetency.pickle', 'wb') as handle:
    pickle.dump(X_test_appetency, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

In [ ]:

```
with open('y_test_appetency.pickle', 'wb') as handle:
    pickle.dump(y_test_appetency, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

In [ ]:

```
with open('train_labels.pickle', 'wb') as handle:
    pickle.dump(train_labels, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

In [ ]:

```
with open('test_labels.pickle', 'wb') as handle:
    pickle.dump(test_labels, handle, protocol=pickle.HIGHEST_PROTOCOL)
```