Machine Learning Project

Missing value imputation analysis on ML1 Dataset

In [1]:
```
# Import libraries
import pandas as pd
import numpy as np
import pyreadstat
import seaborn as sns
import matplotlib.pyplot as plt
import random
import math
from sklearn import preprocessing
from statistics import mode
```

In [2]:
```
# Set path to the file location
path = 'CleanedDataset.sav'
```

In [3]:
```
# Read the data
df, meta = pyreadstat.read_sav(path)
```

In [4]:
```
# Sample of rows present in dataset
df.head()
```

Out[4]:

	session_id	session_date	last_update_date	session_last_update_date	referrer	creation_date	s
0	2400853.0	2013-08-28 12:15:55	8/28/13 12:15	8/28/13 12:15	abington	2013-08-28 11:51:56	
1	2400856.0	2013-08-28 12:13:49	8/28/13 12:13	8/28/13 12:13	abington	2013-08-28 11:52:27	
2	2400860.0	2013-08-28 12:15:57	8/28/13 12:15	8/28/13 12:15	abington	2013-08-28 11:52:58	
3	2400868.0	2013-08-28 12:12:21	8/28/13 12:12	8/28/13 12:12	abington	2013-08-28 11:53:35	
4	2400872.0	2013-08-28 12:11:58	8/28/13 12:11	8/28/13 12:11	abington	2013-08-28 11:54:04	

5 rows × 382 columns

In [5]:
```
# Basic summary statistics
df.describe()
```

Out[5]:

	session_id	age	sample	sunkgroup	sunkDV	gainlossgroup	gainloss
count	6.344000e+03	6328.000000	6344.000000	6344.000000	6330.000000	6344.000000	6271.000
mean	2.436417e+06	25.975980	17.887295	0.486602	7.553555	0.505675	1.519
std	9.493816e+04	11.351214	8.196722	0.499860	2.246900	0.500007	0.499
min	6.196300e+05	12.000000	1.000000	0.000000	1.000000	0.000000	1.000
25%	2.410160e+06	19.000000	13.000000	0.000000	7.000000	0.000000	1.000
50%	2.435912e+06	21.000000	18.000000	0.000000	9.000000	1.000000	2.000
75%	2.477297e+06	28.000000	23.000000	1.000000	9.000000	1.000000	2.000
max	2.511739e+06	100.000000	36.000000	1.000000	9.000000	1.000000	2.000

8 rows × 180 columns

In [6]:
```
# Go through the columns and find the unique values
for col_name in df:
    print(col_name, df[col_name].unique())
    print()
```

```
 'We were given a list of studies we could choose by a professor.'
 '4 participants in the study' '4 participants in this study'
 '12:00 session' '12:40 session' '1:20 session' '2:00 session'
 '2:40 session' '3:20 session' '4:00 session' '4:40 session'
 '12:30 session' '2:30 session' '3:30 session' '1:00 session'
 '1:30 session' '2:30 sessioni' '3:00 session' '9:30 session' '1'
 '3:30 sessioin'
 'Students actively recruited from W&L Psychology Department'
 'Students actively recruited from Washington and Lee Psychology Department.'
 'This study was required for the social psychology class, but was completed
in groups outside of class.'
 'Extra Credit']

numparticipants_actual ['' '16' '14' '18' '17' '20' '21' '24' '22' '23']

numparticipants ['5' '.' '9' '1' '3' '2' '7' '4' '10' '20' '8' '6' '18' '15'
'16' '' '17'
 '13' '11' '14' '12' '19' '21' '22' '23' '0']
```

Basic inferences drawn

- Total number of rows *columns is 6344* 382
- Columns are classified into one of the four categories :
    1. 0 - Text
    2. 1 - Numerical
    3. 2 - Categorical

4. 3 - Bad

In [23]:
```
# Numerical variables
num_var = ['age',
 'anchoring1a',
 'anchoring1b',
 'anchoring2a',
 'anchoring2b',
 'anchoring3a',
 'anchoring3b',
 'anchoring4a',
 'anchoring4b',
 'artwarm',
 'gamblerfallacya',
 'gamblerfallacyb',
 'mathwarm',
 'moneyagea',
 'moneyageb',
 'omdimc3rt',
 'omdimc3trt',
 'anchoring1akm',
 'anchoring3ameter',
 'mturk.total.mini.exps',
 'meanlatency',
 'meanerror',
 'block2_meanerror',
 'block3_meanerror',
 'block5_meanerror',
 'block6_meanerror',
 'lat11',
 'lat12',
 'lat21',
 'lat22',
 'sd1',
 'sd2',
 'd_art1',
 'd_art2',
 'd_art',
 'sunkDV',
 'anchoring1',
 'anchoring2',
 'anchoring3',
 'anchoring4',
 'Ranchori',
 'RAN001',
 'RAN002',
 'RAN003',
 'Ranch1',
 'Ranch2',
 'Ranch3',
 'Ranch4',
 'gambfalDV',
 'quotearec',
 'quotebrec',
 'totalflagestimations',
 'totalnoflagtimeestimations',
 'flagdv',
 'Sysjust',
```

```
 'Imagineddv',
 'IATexpart',
 'IATexpmath',
 'IATexp.overall',
 'totexpmissed']

cat_var = ['referrer',
 'expgender',
 'exprace',
 'exprunafter',
 'exprunafter2',
 'compensation',
 'recruitment',
 'separatedornot',
 'allowedforbiddena',
 'allowedforbiddenb',
 'citizenship',
 'diseaseframinga',
 'diseaseframingb',
 'ethnicity',
 'flagdv1',
 'flagdv2',
 'flagdv3',
 'flagdv4',
 'flagdv5',
 'flagdv6',
 'flagdv7',
 'flagdv8',
 'flagsupplement1',
 'flagsupplement2',
 'flagsupplement3',
 'flagtimeestimate1',
 'flagtimeestimate2',
 'flagtimeestimate3',
 'flagtimeestimate4',
 'iatexplicitart1',
 'iatexplicitart2',
 'iatexplicitart3',
 'iatexplicitart4',
 'iatexplicitart5',
 'iatexplicitart6',
 'iatexplicitmath1',
 'iatexplicitmath2',
 'iatexplicitmath3',
 'iatexplicitmath4',
 'iatexplicitmath5',
 'iatexplicitmath6',
 'imaginedexplicit1',
 'imaginedexplicit2',
 'imaginedexplicit3',
 'imaginedexplicit4',
 'major',
 'moneygendera',
 'moneygenderb',
 'nativelang',
 'nativelang2',
 'noflagtimeestimate1',
```

```
'noflagtimeestimate2',
'noflagtimeestimate3',
'noflagtimeestimate4',
'omdimc3',
'politicalid',
'quotea',
'quoteb',
'race',
'reciprocityothera',
'reciprocityotherb',
'reciprocityusa',
'reciprocityusb',
'scalesa',
'scalesb',
'sex',
'sunkcosta',
'sunkcostb',
'sysjust1',
'sysjust2',
'sysjust3',
'sysjust4',
'sysjust5',
'sysjust6',
'sysjust7',
'sysjust8',
'previous_session_schema',
'us_or_international',
'lab_or_online',
'religion',
'priorexposure1',
'priorexposure10',
'priorexposure11',
'priorexposure12',
'priorexposure13',
'priorexposure2',
'priorexposure3',
'priorexposure4',
'priorexposure5',
'priorexposure6',
'priorexposure7',
'priorexposure8',
'priorexposure9',
'mturk.non.US',
'mturk.Submitted.PaymentReq',
'mturk.duplicate',
'mturk.exclude.null',
'mturk.keep',
'filter_$',
'order',
'iat_exclude',
'o1',
'o2',
'o3',
'o4',
'o5',
'o6',
'o7',
```

```
 'o8',
 'o9',
 'o10',
 'o11',
 'scalesorder',
 'reciprocorder',
 'diseaseforder',
 'quoteorder',
 'flagprimorder',
 'sunkcostorder',
 'anchorinorder',
 'allowedforder',
 'gamblerforder',
 'moneypriorder',
 'imaginedorder',
 'sample',
 'sunkgroup',
 'gainlossgroup',
 'gainlossDV',
 'anch1group',
 'anch2group',
 'anch3group',
 'anch4group',
 'gambfalgroup',
 'scalesgroup',
 'scalesreca',
 'scalesrecb',
 'scales',
 'reciprocitygroup',
 'reciprocityother',
 'reciprocityus',
 'allowedforbiddenGroup',
 'allowedforbidden',
 'quoteGroup',
 'flagfilter',
 'flagGroup',
 'MoneyGroup',
 'moneyfilter',
 'ContactGroup',
 'IATfilter',
 'partgender',
 'IATEXPfilter']

bad_var = ['session_id',
 'session_date',
 'last_update_date',
 'session_last_update_date',
 'creation_date',
 'session_creation_date',
 'numparticipants_actual',
 'numparticipants',
 'imptaskto',
 'user_id',
 'session_status',
 'previous_session_id',
 'mturk_worker_id',
 'pi_referrer',
```

```
'user_agent',
'task_status',
'task_sequence',
'session_created_by',
'study_url',
'study_name',
'task_id.0',
'task_id.1',
'task_id.2',
'task_id.3',
'task_id.4',
'task_id.5',
'task_id.6',
'task_id.7',
'task_id.8',
'task_id.9',
'task_id.10',
'task_id.11',
'task_id.12',
'task_id.13',
'task_id.14',
'task_id.15',
'task_id.16',
'task_id.17',
'task_id.18',
'task_id.19',
'task_id.20',
'task_id.21',
'task_id.22',
'task_id.23',
'task_id.24',
'task_id.25',
'task_id.26',
'task_id.27',
'task_id.28',
'task_id.29',
'task_id.30',
'task_id.31',
'task_id.32',
'task_id.33',
'task_id.34',
'task_id.35',
'task_id.36',
'task_id.37',
'task_id.38',
'task_id.39',
'task_id.40',
'task_id.41',
'task_id.42',
'task_id.43',
'task_id.44',
'task_url.0',
'task_url.1',
'task_url.2',
'task_url.3',
'task_url.4',
'task_url.5',
```

```
'task_url.6',
'task_url.7',
'task_url.8',
'task_url.9',
'task_url.10',
'task_url.11',
'task_url.12',
'task_url.13',
'task_url.14',
'task_url.15',
'task_url.16',
'task_url.17',
'task_url.18',
'task_url.19',
'task_url.20',
'task_url.21',
'task_url.22',
'task_url.23',
'task_url.24',
'task_url.25',
'task_url.26',
'task_url.27',
'task_url.28',
'task_url.29',
'task_url.30',
'task_url.31',
'task_url.32',
'task_url.33',
'task_url.34',
'task_url.35',
'task_url.36',
'task_url.37',
'task_url.38',
'task_url.39',
'task_url.40',
'task_url.41',
'task_url.42',
'task_url.43',
'task_url.44',
'task_creation_date.0',
'task_creation_date.1',
'task_creation_date.2',
'task_creation_date.3',
'task_creation_date.4',
'task_creation_date.5',
'task_creation_date.6',
'task_creation_date.7',
'task_creation_date.8',
'task_creation_date.9',
'task_creation_date.10',
'task_creation_date.11',
'task_creation_date.12',
'task_creation_date.13',
'task_creation_date.14',
'task_creation_date.15',
'task_creation_date.16',
'task_creation_date.17',
```

```
 'task_creation_date.18',
 'task_creation_date.19',
 'task_creation_date.20',
 'task_creation_date.21',
 'task_creation_date.22',
 'task_creation_date.23',
 'task_creation_date.24',
 'task_creation_date.25',
 'task_creation_date.26',
 'task_creation_date.27',
 'task_creation_date.28',
 'task_creation_date.29',
 'task_creation_date.30',
 'task_creation_date.31',
 'task_creation_date.32',
 'task_creation_date.33',
 'task_creation_date.34',
 'task_creation_date.35',
 'task_creation_date.36',
 'task_creation_date.37',
 'task_creation_date.38',
 'task_creation_date.39',
 'task_creation_date.40',
 'task_creation_date.41',
 'task_creation_date.42',
 'task_creation_date.43',
 'task_creation_date.44',
 'task_id.45',
 'task_url.45',
 'task_creation_date.45',
 'beginlocaltime',
 'gamblerfallacya_sd',
 'gamblerfallacyb_sd',
 'iatorder',
 'anchoring1bkm',
 'anchoring3bmeter',
 'citizenship2',
 'mturk.exclude']

text_var = ['expcomments', 'feedback', 'imagineddescribe', 'text', 'moneyethnicit
```

In [24]:
```
# Adding the count of variables across the categories to see if it adds up to 382
len(num_var)+len(cat_var)+len(bad_var)+len(text_var)
```

Out[24]: 382

382 variables have been classified into four categories.

In [25]:
```
# Remove the values '', '.', 'null' and 'NaT' in all the columns
df = df.replace('', np.nan)
df = df.replace('.', np.nan)
df = df.replace('null', np.nan)
df = df.replace('NaT', np.nan)
df = df.replace('n/a', np.nan)
df = df.replace('N/A', np.nan)
```

In [26]:
```
# Checking after replacing with Nan's
for col_name in df:
    print(col_name, df[col_name].unique())
    print()
```

```
 'subject ID is 96' 'subject ID is 97' 'subject ID is 98'
 'subject ID is 99' 'subject ID is 100' 'subject ID is 101'
 'subject ID is 102' 'ID: 101' 'ID: 102' 'ID: 103' 'ID: 104' 'ID: 105'
 'ID: 106' 'ID: 107' 'ID: 108' 'ID: 109' 'ID: 110' 'ID: 111' 'ID: 113'
 'ID: 114' 'ID: 116' 'ID: 115' 'ID: 112' 'ID: 117' 'ID: 118' 'ID: 119'
 'ID: 120' 'ID: 122' 'ID: 121' 'ID: 123' 'ID: 124' 'ID: 125' 'ID: 126'
 'ID: 127' 'ID: 128' 'ID: 129' 'ID: 130' 'ID: 131' 'ID: 132' 'ID: 133'
 '134' '136' '135' '137' '138' '139' '140' '141' '142' '143' '144' '145'
 '146' '147' 'ID: 148' 'ID: 149' 'ID: 150' 'ID: 151' 'ID: 152'
 'I signed up for this study via the website provided by by my psychology pro
fessor.'
 'ID: 154' 'ID: 155' 'ID: 156' 'ID: 157' 'ID: 158' 'ID: 159' 'ID: 160'
 'ID: 161' 'ID:162' 'ID:163' 'ID:164' 'ID;165' 'ID:166' 'ID167' 'ID:173'
 'ID:172' 'ID:168' 'ID:169' 'ID:170' 'ID:171' '180' '185' '181' '182'
 '183' 'ID: 186' 'ID: 187' 'ID: 188' 'ID: 175' 'ID: 176' 'ID: 177'
 'ID: 178' '174' 'ID: 179' 'ID: 184' 'ID: 189' 'ID: 190' 'ID: 191'
 'ID: 192' 'ID: 193' 'ID: 194' 'ID: 195' 'ID: 196' 'ID: 197' 'ID: 198'
 'ID: 199' 'ID: 200' 'ID: 201' 'ID: 202' 'ID: 203' '204' '205' '206' '207'
 '208' '209' '210' '211' '212' '213' '214' 'ID: 221' 'ID: 222' 'ID: 223'
 'ID: 224' 'ID: 225' 'ID: 226' 'ID: 227' 'ID: 229' 'ID: 230' 'ID: 231'
```

In [27]:
```
# Considering variables that are present only in numerical and categorical variab
print("The total number of numerical columns present are ", len(num_var))
print("The total number of categorical columns present are ", len(cat_var))
print("The total number of textual columns present are ", len(text_var))
print("The total number of bad columns present are ", len(bad_var))
```

```
The total number of numerical columns present are  60
The total number of categorical columns present are  150
The total number of textual columns present are  6
The total number of bad columns present are  166
```

In [28]:
```
print("The total number of useful columns are ", len(num_var)+len(cat_var))
```

```
The total number of useful columns are  210
```

In [29]:
```
new_df = pd.DataFrame()
```

In [30]:
```
# Create a new df with only selected columns
new_df = pd.DataFrame()

for col_name in df:
    if(col_name in num_var or col_name in cat_var):
        new_df[col_name] = df[col_name]
```

In [31]:
```
# Check the total number of columns present
new_df.count()
```

Out[31]:
```
referrer           6344
expgender          2945
exprace            2945
exprunafter        2979
exprunafter2        721
compensation       2980
recruitment        2983
separatedornot     2946
age                6328
sample             6344
sunkgroup          6344
sunkDV             6330
gainlossgroup      6344
gainlossDV         6271
anch1group         6344
anch2group         6344
anch3group         6344
anch4group         6344
anchoring1         5362
anchoring2         5284
anchoring3         5627
anchoring4         5609
Ranchori           5362
RAN001             5284
RAN002             5627
RAN003             5609
Ranch1             5362
Ranch2             5284
Ranch3             5627
Ranch4             5609
                   ...
lat21              6234
lat22              6257
sd1                6250
sd2                6258
d_art1             6220
d_art2             6213
d_art              6185
iat_exclude        6344
o1                 6344
o2                 6344
o3                 6344
o4                 6344
o5                 6344
o6                 6344
o7                 6344
o8                 6344
o9                 6344
o10                6344
o11                6344
scalesorder        6344
reciprocorder      6344
diseaseforder      6344
```

```
quoteorder         6344
flagprimorder      6344
sunkcostorder      6344
anchorinorder      6344
allowedforder      6344
gamblerforder      6344
moneypriorder      6344
imaginedorder      6344
Length: 210, dtype: int64
```

In [32]:
```
# Checking all the unique values from each column in order to clean
for col_name in new_df:
    if(col_name in cat_var):
        print(col_name, new_df[col_name].unique())
        print()
```

```
flagdv2 [ 1.  2.  4.  6.  3.  7.  5. nan]

flagdv3 [ 1.  4.  7.  3.  5.  2.  6. nan]

flagdv4 [ 1.  4.  3.  6.  2.  7.  5. nan]

flagdv5 [ 4.  3.  2.  1. nan  5.  6.  7.]

flagdv6 [ 7.  1.  3.  6.  5.  4.  2. nan]

flagdv7 [ 4.  6.  5.  2.  7.  1. nan  3.]

flagdv8 [ 4.  2.  1.  3.  7.  5.  6. nan]

flagsupplement1 [11.  6.  4.  8.  7.  5.  1.  9. 10.  3.  2. nan]

flagsupplement2 [ 4.  7.  5.  3.  1.  2. nan  6.]

flagsupplement3 [ 4.  5.  3.  6.  7.  2.  1. nan]
```

In [33]:
```
len(new_df.dtypes[new_df.dtypes == object])
```

Out[33]: 40

In [34]:
```
len(cat_var)
```

Out[34]: 150

Columns to clean are :

- exprunafter2 - Convert all to lower
- nativelang2 - Convert all to lower

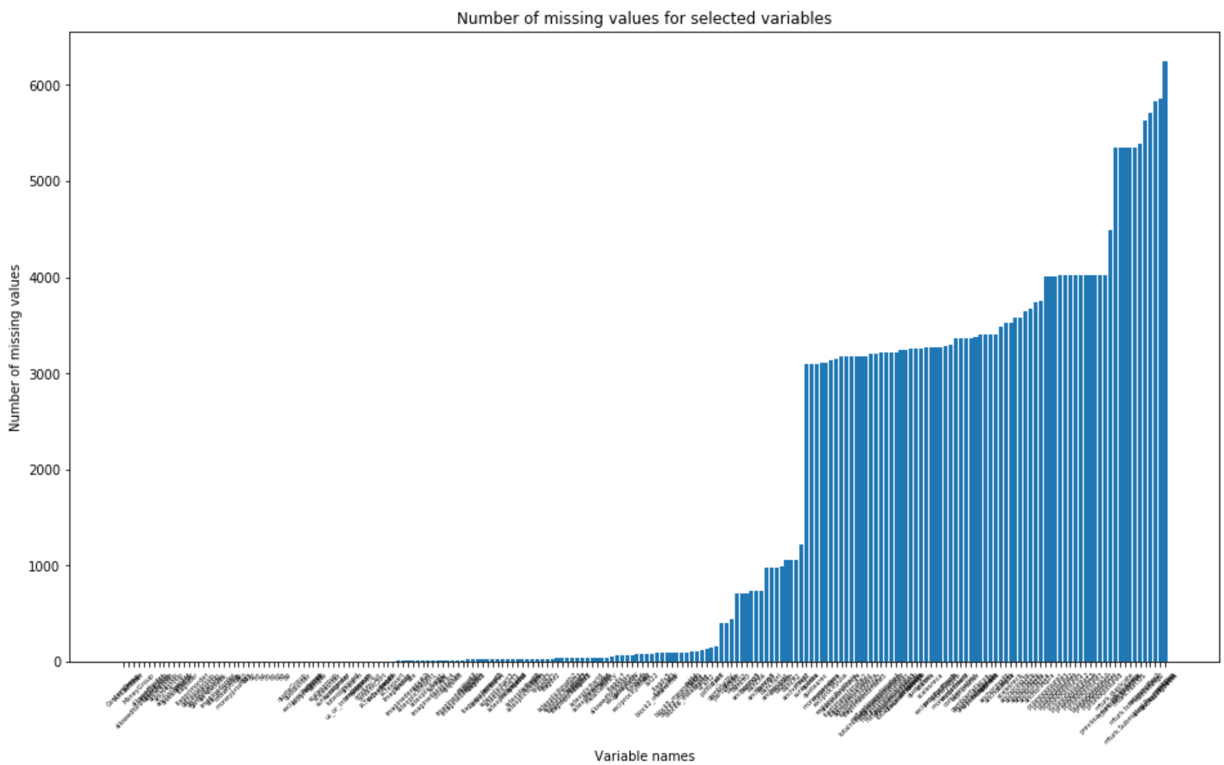
In [35]:
```
# Clean the columns
new_df['exprunafter2'] = new_df['exprunafter2'].str.lower()
new_df['nativelang2'] = new_df['nativelang2'].str.lower()
```

In [36]:
```
# Computing the total number of Nans for each column
no_of_nans = {}

for col_name in new_df:
        no_of_nans[col_name] = sum(pd.isna(new_df[col_name]))
print("The total number of columns considered are ", len(no_of_nans))
```

```
The total number of columns considered are  210
```

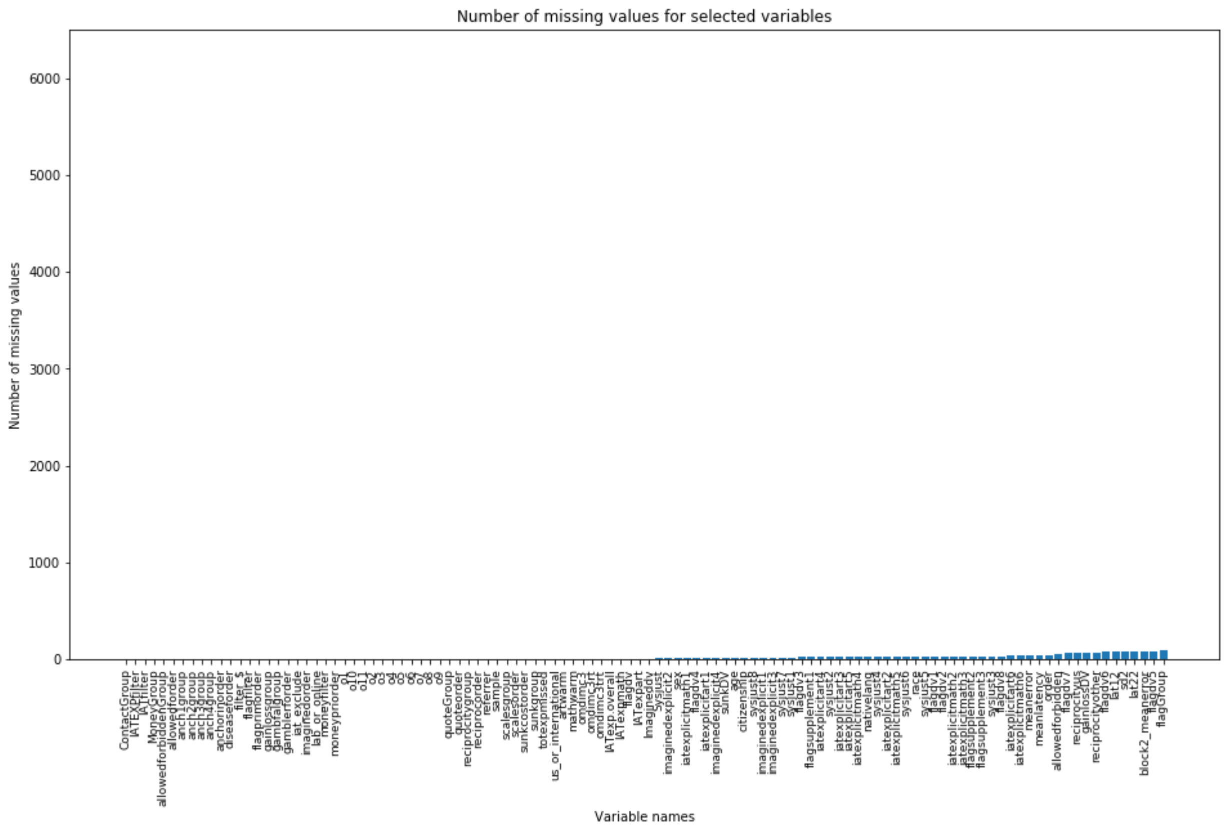
In [37]:
```
# After sorting the variables
no_of_nans_sorted = sorted(no_of_nans.items(), key= lambda kv:(kv[1], kv[0]))
```

In [38]:
```
# Plot the variables and then plot them
x_val = [x[0] for x in no_of_nans_sorted]
y_val = [y[1] for y in no_of_nans_sorted]

plt.figure(figsize=(16,9))
plt.bar(x_val, y_val)
plt.xticks(fontsize = 5, rotation='45')
plt.xlabel('Variable names')
plt.ylabel('Number of missing values')
plt.title('Number of missing values for selected variables')
plt.show()
```

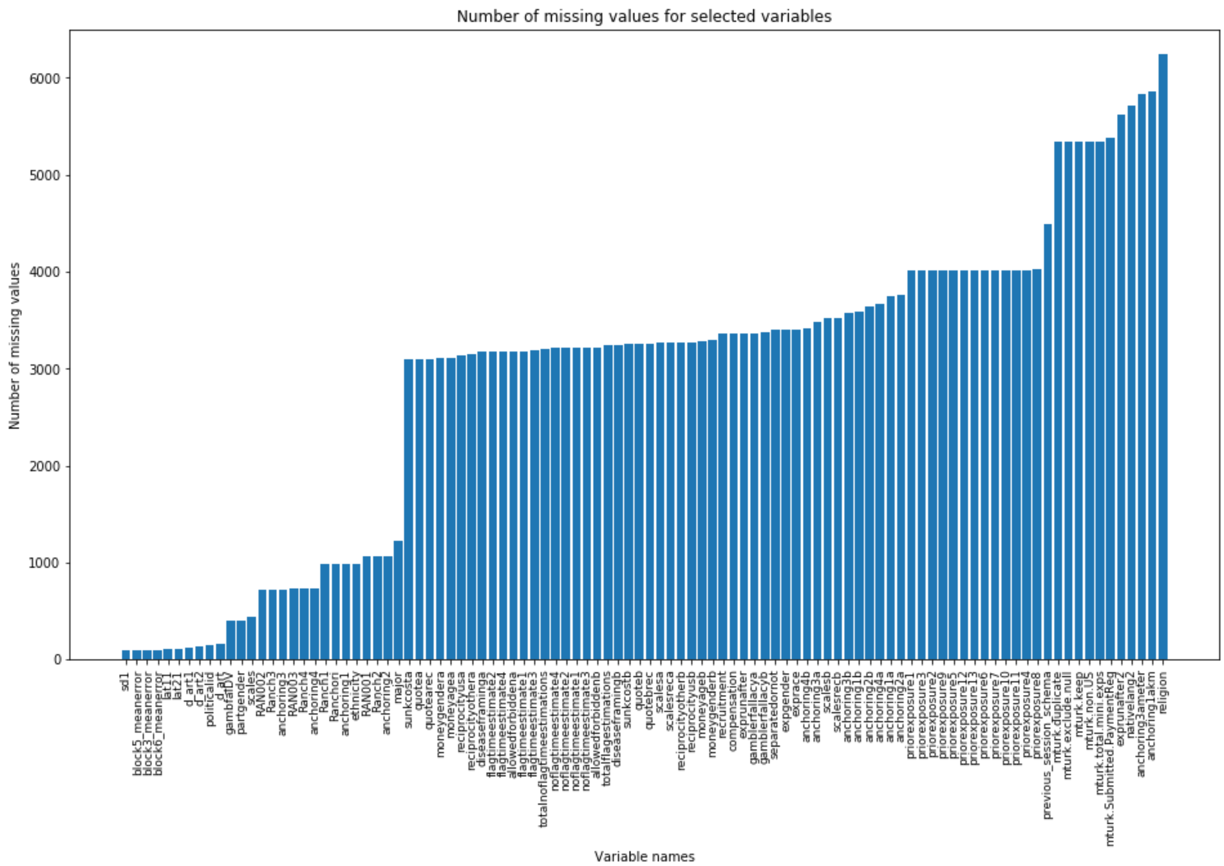


Note that the above graph is split as two graphs and plotted below.

In [39]:
```python
# First 110
plt.figure(figsize=(16,9))
plt.bar(x_val[0:110], y_val[0:110])
plt.xticks(fontsize = 9, rotation='90')
plt.xlabel('Variable names')
plt.ylim(0,6500)
plt.ylabel('Number of missing values')
plt.title('Number of missing values for selected variables')
plt.show()
```



In [40]:
```
# Remaining
plt.figure(figsize=(16,9))
plt.bar(x_val[110:], y_val[110:])
plt.xticks(fontsize = 9, rotation='90')
plt.xlabel('Variable names')
plt.ylim(0,6500)
plt.ylabel('Number of missing values')
plt.title('Number of missing values for selected variables')
plt.show()
```



In [41]:
```
categorical_unique_counts = {}
for i in cat_var:
    categorical_unique_counts[i] = len(new_df[i].unique())
```

In [42]:
```
categorical_unique_counts
```

```
 'imaginedorder': 11,
 'sample': 36,
 'sunkgroup': 2,
 'gainlossgroup': 2,
 'gainlossDV': 3,
 'anch1group': 2,
 'anch2group': 2,
 'anch3group': 2,
 'anch4group': 2,
 'gambfalgroup': 2,
 'scalesgroup': 2,
 'scalesreca': 3,
 'scalesrecb': 3,
 'scales': 3,
 'reciprocitygroup': 2,
 'reciprocityother': 3,
 'reciprocityus': 3,
 'allowedforbiddenGroup': 2,
 'allowedforbidden': 3,
 'quoteGroup': 2,
```

In [43]:
```
new_df['nativelang'].unique()
```

Out[43]:
```
array(['english', 'other', 'spanish', nan, 'portuguese', 'czech',
       'slovak', 'malay', 'turkish', 'polish', 'dutch', 'italian'],
      dtype=object)
```

In [44]:
```
no_of_nans['nativelang2']
```

Out[44]:
```
5711
```

In [45]:
```
# Things to be removed.
'nativelang2','exprunafter2'
```

Out[45]:
```
('nativelang2', 'exprunafter2')
```

In [46]:
```
no_of_nans['exprunafter2']
```

Out[46]:
```
5623
```

Imputing Numeric column value by Mean and Categorical by Mode

In [90]:
```
def estimate_error_mean_mode(new_df = new_df ,cat_var = cat_var, num_var = num_va
    num_error = {}
    cat_error = {}
    for i in num_var:
        if missing_ratio is None:
            missing_ratio = no_of_nans[i]/new_df.shape[0]
        #Removing missing data
        data = new_df[i].dropna().reset_index(drop = True)
        #Scaling the data
        minimum = min(data)
        maximum = max(data)
        transformed = (data - minimum) / (maximum - minimum)
        org = transformed.copy()
        error_temp = []
        #Doings folds to get average error
        for j in range(folds):
            sample_index = random.sample(list(data.index), math.floor(missing_rat
            transformed[sample_index] = np.nan
            transformed = transformed.fillna(transformed.mean())
            error_temp.append(np.sum((org - transformed)**2)/len(sample_index))
        num_error[i] = sum(error_temp)/folds

    for i in cat_var:
        if missing_ratio is None:
            missing_ratio = no_of_nans[i]/new_df.shape[0]
        #Removing missing data
        data = new_df[i].dropna().reset_index(drop = True)
        org = data.copy()
        error_temp = []
        #Doings folds to get average error
        for j in range(folds):
            sample_index = random.sample(list(data.index), math.floor(missing_rat
            data[sample_index] = np.nan
            data = data.fillna(data.mode())
            error_temp.append(np.mean(org != data))
        cat_error[i] = sum(error_temp)/folds

    print("Numeric attributes imputation error : ", sum(num_error.values()))
    print("Categories attributes imputation error : ", sum(cat_error.values()))
    return num_error, cat_error
```

In [54]:
```
numerical_error, categorical_error = estimate_error_mean_mode()
```

```
Numeric attributes imputation error :  6.444154820703175
Categories attributes imputation error :  39.39871892366889
```

Plotting the Errors obtained

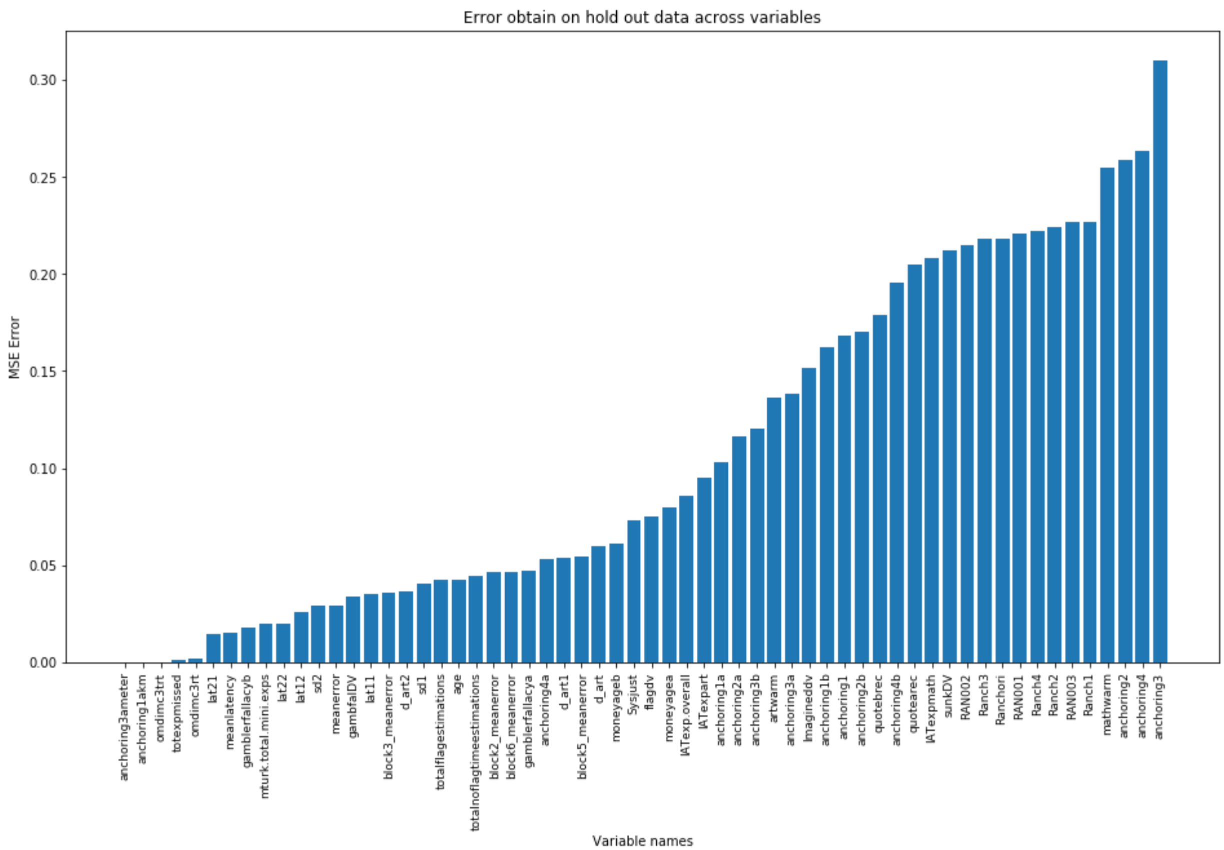
In [83]:
```
import operator

sorted_num = sorted(numerical_error.items(), key=operator.itemgetter(1))
sorted_cat = sorted(categorical_error.items(), key=operator.itemgetter(1))
```

In [84]:
```
x = [x[0] for x in sorted_num]
y = [x[1] for x in sorted_num]
```

In [85]:
```
sum(y[:-4])
```

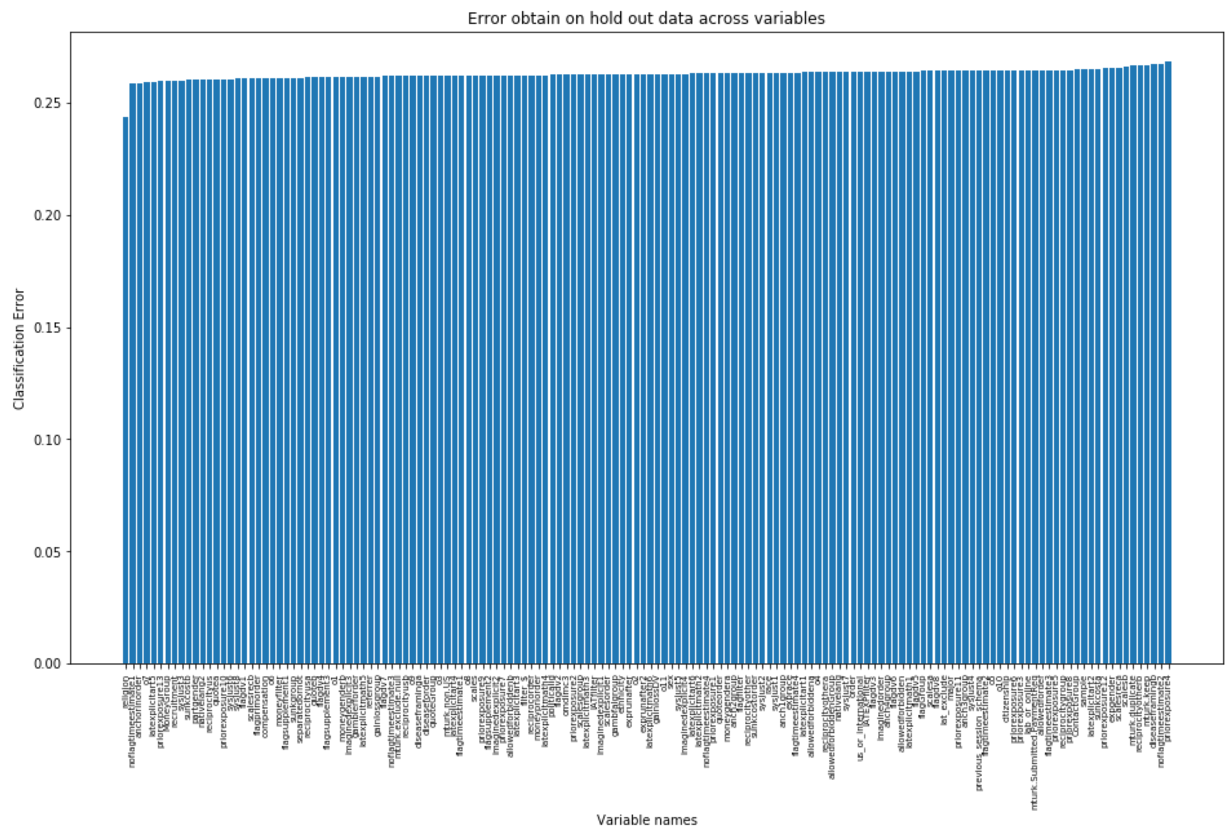
Out[85]:
```
5.357539899844292
```

In [72]:
```
plt.figure(figsize=(16,9))
plt.bar(x, y)
plt.xticks(fontsize = 9, rotation='90')
plt.xlabel('Variable names')
plt.ylabel('MSE Error')
plt.title('Error obtain on hold out data across variables')
plt.show()
```



In [75]:
```
x = [x[0] for x in sorted_cat]
y = [x[1] for x in sorted_cat]
```

In [76]:
```
plt.figure(figsize=(16,9))
plt.bar(x, y)
plt.xticks(fontsize = 7, rotation='90')
plt.xlabel('Variable names')
plt.ylabel('Classification Error')
plt.title('Error obtain on hold out data across variables')
plt.show()
```



Imputing using EM Algorithm

In [84]:
```
# Create a df with only numerical variables from new_df
num_df = new_df.loc[:, num_var].copy()
```

In [85]:
```
# Check the length of the numerical dataframe
print("The length of the numerical dataframe is ", len(num_df.columns))
print("The total number of numerical dataframe are ", len(num_df))
```

```
The length of the numerical dataframe is  60
The total number of numerical dataframe are  6344
```

In [90]:
```
sum(num_df.isna().sum(axis = 1) == 0)
```

Out[90]: 0

In [124]:
```
def EM(data, loops = 50):

    nulls_present = np.argwhere(np.isnan(data))

    # For each row and column in the null columns
    for row in nulls_present:

        cur_col = data[:]

        # Take the values that are present for a column and

        # Compute the mean and sd of the column selected without the missing valu
        mu = cur_col[~np.isnan(cur_col)].mean()
        std = cur_col[~np.isnan(cur_col)].std()

        # Fill the missing values with a random distribution - with the computed 
        cur_col[row] = np.random.normal(loc=mu, scale=std)

        prev, i = 1, 1

        for i in range(loops):
            # Expectation
            # Recompute the mean and sd after replacing the new missing value
            mu = cur_col[~np.isnan(cur_col)].mean()
            std = cur_col[~np.isnan(cur_col)].std()

            # Maximization
            # Fill the missing value again with the newly estimated mean and sd
            cur_col[row] = np.random.normal(loc=mu, scale=std)

            # If likelihood doesn't change by atleast 10% the loop breaks
            # Min number of runs = 5
            delta_val = (cur_col[row] - prev)/prev

            if (i > 5 and delta_val.item() < 0.1):
                data[row] = cur_col[row]
                break
            data[row] = cur_col[row]
            prev = cur_col[row]

    return data
```

Note - The input array needs to be converted to a numpy array before being sent as an input.

In [134]:
```
def estimate_error_EM(new_df = num_df , num_var = num_var, folds = 5, missing_rat
    num_error = {}
    for i in num_var:
        if missing_ratio is None:
            missing_ratio = no_of_nans[i]/new_df.shape[0]
        data = new_df[i].dropna().reset_index(drop = True)
        minimum = min(data)
        maximum = max(data)
        transformed = (data - minimum) / (maximum - minimum)
        org = transformed.copy()
        error_temp = []
        for j in range(folds):
            sample_index = random.sample(list(data.index), math.floor(missing_rat
            transformed[sample_index] = np.nan
            transformed = EM(transformed, loops= 50)
            error_temp.append(np.sum((org - transformed)**2)/len(sample_index))
        num_error[i] = sum(error_temp)/folds
    return num_error
```

In [135]:
```
num_error_em = estimate_error_EM()
```

In [142]:
```
em_error = pd.Series(num_error_em)
```

In [143]:
```
em_error.to_csv("EM_Error.csv")
```

In [136]:
```
print("Numeric attributes imputation error : ", sum(num_error_em.values()))
```

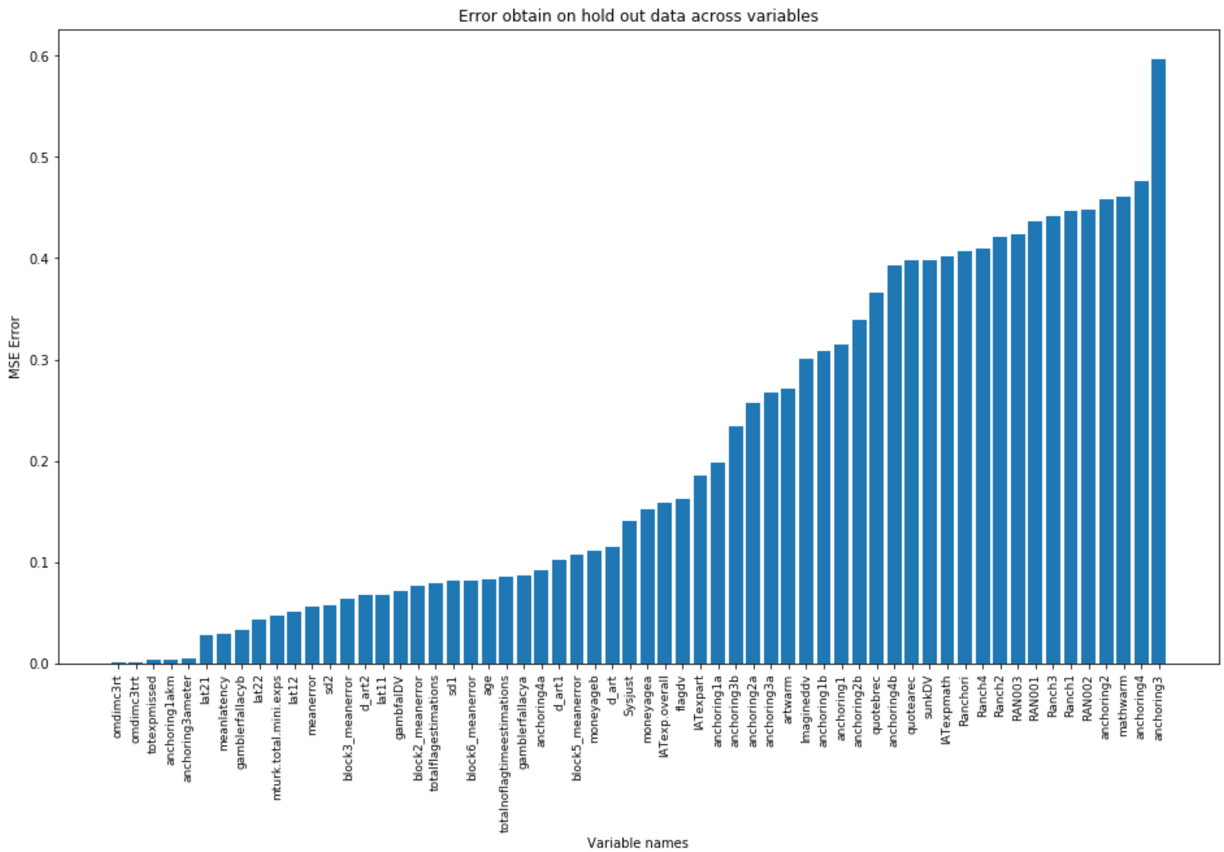
```
Numeric attributes imputation error :  12.411369724301334
```

Plotting the Errors obtained

In [137]:
```
import operator

sorted_num = sorted(num_error_em.items(), key=operator.itemgetter(1))
```

In [144]:
```
x = [x[0] for x in sorted_num]
y = [x[1] for x in sorted_num]
```

In [140]:
```
plt.figure(figsize=(16,9))
plt.bar(x, y)
plt.xticks(fontsize = 9, rotation='90')
plt.xlabel('Variable names')
plt.ylabel('MSE Error')
plt.title('Error obtain on hold out data across variables')
plt.show()
```



Imputing Numeric column value by Linear Regression and Categorical by Logistic

Setting up functions that will be helpful in the pipeline.

In [30]:
```
#Function to get filled data
def get_filled_columns(data):
    curr_missing = data.isna().sum()
    non_missing = curr_missing[curr_missing == 0]
    return list(non_missing.index)
```

In [31]:
```
#Function to get column with least missing values
def get_least_missing_column(data):
    curr_missing = data.isna().sum()
    missing = curr_missing[curr_missing != 0]
    return (missing.sort_values().index[0], missing.sort_values()[0])
```

In [32]:
```
#KFold split performation
def kFold(input_data, k = 10):
    temp = list(range(input_data.shape[0]))
    random.shuffle(temp)
    per_fold = math.floor(len(temp)/folds)
    folds_data= {}
    for i in range(folds):
        data = input_data
        start = i * per_fold
        end = (i + 1) * per_fold
        if i == folds - 1:
            folds_data[i] = {}
            folds_data[i]['Val_data'] = data.iloc[temp[start:]]
            folds_data[i]['Train_data'] = data.drop(temp[start:], axis = 0)
        else:
            folds_data[i] = {}
            folds_data[i]['Val_data'] = data.iloc[temp[start:end]]
            folds_data[i]['Train_data'] = data.drop(temp[start:end], axis = 0)
            val_data = data.iloc[temp[start:end]]
    return folds_data
```

In [33]:
```
#Min Max scaler. Transform the data into 0-1.
class min_max_scaler():
    def __init__ (self, data, y = None):
        self.data = data
        self.y = y
        self.min_params = {}
        self.max_params = {}
        self.single = {}
        for i in data.columns:
            if (i != self.y) :
                if (len(data[i].unique())>1):
                    self.min_params[i] = min(data[i])
                    self.max_params[i] = max(data[i])
                else:
                    self.single[i] = 0.001

    def transform(self, data):
        copy = data.copy()
        for i in data.columns:
            if ((i != self.y) and (i not in self.single.keys())):
                copy[i] = data[i].apply(lambda x : (x- self.min_params[i]) / (sel
            elif (i in self.single.keys()):
                copy[i] = self.single[i]
            else:
                copy[i] = data[i]
        return copy
```

Linear Regression

In [34]:
```
class LinearRegression():

    def __init__(self, method = None, lambda_value = 0.1):
        self.method = method
        self.lambda_value = lambda_value

    def prepare_data(self, data, target):
        data['Bias'] = 1
        self.variables = data.drop(target, axis = 1).columns
        self.X = data.drop(target, axis = 1).values
        data.drop('Bias', axis = 1, inplace = True)
        self.Y = data[target].values

    def fit(self, data, target):
        self.data = data
        self.target = target
        self.prepare_data(self.data, self.target)

        if self.method == None :
            self.weights = np.matmul(np.linalg.inv(np.matmul(self.X.T, self.X)),
        elif self.method == "Ridge" :
            #print(self.X.T)
            #print(np.matmul(self.X.T, self.X))
            self.weights = np.matmul(np.linalg.inv(np.matmul(self.X.T, self.X) +
        elif self.method == "Lasso":
            #print("Working..")
            count_weight = self.X.shape[1]
            self.weights = [0 for i in range(count_weight)]
            while True:
                old_weights = self.weights.copy()
                for i in range(len(self.weights)):
                    denom_value = np.matmul(self.X[:,i].T, self.X[:,i])
                    actual_value = (self.Y - np.matmul(self.X,self.weights))
                    cal_x_upper = (np.matmul((-1 * self.X[:,i].T), actual_value)
                    cal_x_lower = (np.matmul((-1 * self.X[:,i].T), actual_value)
                    if  cal_x_upper < self.weights[i] :
                        self.weights[i] = self.weights[i] + (np.matmul((self.X[:,
                    elif cal_x_lower > self.weights[i] :
                        self.weights[i] = self.weights[i] +(np.matmul((self.X[:,i
                    else:
                        self.weights[i] = 0
                #Stopping criteria
                updates = [k - l for k, l in zip(old_weights, self.weights)]
                if max(updates) < 1e-2 and abs(min(updates)) < 1e-2:
                    break


    def predict_row(self, row):
        y_pred = np.sum(np.multiply(self.weights, row))
        return y_pred

    def predict(self,test):
        test['bias'] = 1
        y_predicted = []
        for index,row in test.iterrows():
            y_predicted.append(self.predict_row(row))
```

```
        return y_predicted

    def training_error(self):
        predicted_y = self.predict(self.data.drop(self.target, axis = 1))
        mse = []
        for i in range(len(predicted_y)):
            err = ((predicted_y[i] - self.Y[i])**2)
            mse.append(err)
        return sum(mse)/len(mse)

    def error(self, test):
        test = test.reset_index(drop = True)
        predicted_y = self.predict(test.drop(self.target, axis = 1))
        mse = []
        for i in range(len(predicted_y)):
            err = ((predicted_y[i] - test[self.target][i])**2)
            mse.append(err)
        return sum(mse)/len(mse)
```

Logistic Regression

In [35]:
```
class Logistic_Regression():

    def __init__(self, X, Y, no_of_epochs=1000, learning_rate=0.001, intercept=Tr
        self.X = X
        self.Y = Y
        self.W = None
        self.intercept = intercept
        self.epochs = no_of_epochs
        self.lr = learning_rate
        self.verbose = verbose

    def _add_intercept(self, X):
        intercept = np.ones((X.shape[0],1))
        return np.concatenate((intercept, X), axis=1)

    def _sigmoid_function(self, X):
        return  1 / (1+np.exp(-X))

    def _loss_function(self, X, Y):
        return (-Y * np.log(X) - (1-Y) * np.log(1-X) ).mean()

    def _predict_probs(self, X):
        if(self.intercept):
            X = self._add_intercept(X)
        return (self._sigmoid_function(np.dot(X,self.W)))

    def predict(self, X):
        return self._predict_probs(X)>=0.5

    def error(self, X, y):
        preds = self.predict(X)
        return (preds != y).mean()

    def accuracy(self, X, y):
        preds = self.predict(X)
        return (preds == y).mean()

    def fit(self):
        if(self.intercept):
            self.X = self._add_intercept(self.X)

        self.W = np.zeros(self.X.shape[1])

        iterations = 0

        while(iterations<self.epochs):
            iterations += 1
            pred = self._sigmoid_function(np.dot(self.X, self.W))
            diff = (pred - self.Y)
            self.W -= (self.lr * (np.dot(self.X.T, diff)/self.Y.shape))

            predicted = self._sigmoid_function(np.dot(self.X, self.W))
            cost = self._loss_function(predicted, self.Y)
            if(self.verbose):
                print("Cost: ", cost)
#          print("Cost: ", cost)
```

In [41]:
```
class MultiClass_Logistic_Regression():

    def __init__(self, X, Y, intercept=True, no_of_epochs=1000, learning_rate=0.1
        self.X = X
        self.Y = Y
        self.W = None
        self.intercept = intercept
        self.epochs = no_of_epochs
        self.lr = learning_rate
        self.verbose = verbose
        self.batch_size = batch_size

    def _add_intercept(self, X):
        intercept = np.ones((X.shape[0],1))
        return (np.concatenate((intercept,X), axis=1))

    def _y_one_hot_encode(self):
        self.Y = (np.arange(np.max(self.Y) + 1) == self.Y[:, None]).astype(float)

    def _softmax_function(self, X):
        z = X
        e_x = np.exp(z)
        out = e_x / (1 + e_x.sum(axis = 1, keepdims = True))
#         print(out.sum(axis=1))
        return out
        # To avoid overflow
#         X = X - np.max(X)
#         return (np.exp(X).T/np.sum(np.exp(X),axis=1)).T

    def _loss_function(self, X, Y):
        return (- np.sum(Y * np.log(X), axis=1))

    def _predict_prob(self, X):
        if(self.intercept):
            X = self._add_intercept(X)
        return (self._softmax_function(np.dot(X,self.W)))

    def predict(self, X):
        return np.argmax(self._predict_prob(X), axis=1)

    def error(self, X, y):
        preds = self.predict(X)
        return (preds != y).mean()

    def accuracy(self, X, y):
        preds = self.predict(X)
        return (preds == y).mean()

    def fit(self):
        if(self.intercept):
            self.X = self._add_intercept(self.X)

        self._y_one_hot_encode()
        self.W = np.zeros((self.X.shape[1], self.Y.shape[1]))

        iterations = 0
```

```
        while(iterations < self.epochs):
            iterations += 1

            for i in range(0, self.X.shape[0], self.batch_size):
                x_batch = self.X[i:i+self.batch_size]
                y_batch = self.Y[i:i+self.batch_size]

                z = np.dot(x_batch, self.W)
                pred = self._softmax_function(z)
                diff = (pred - y_batch)

                self.W -= (self.lr * (np.dot(x_batch.T, diff)))
#                 print(self.W)
```

Pipeline for the logistic and linear regression

In [92]:
```
# Initiation and parameters
#num_error = {}
#cat_error = {}
#data = new_df.copy()
hold_out_ratio = 0.1
folds = 5
param1 = ['Ridge']
param2 = [0.1, 0.25, 0.5, 0.75, 1, 1.5, 2, 5]
param3 = [30,50,100,150,200,500,1000]
param4 = [0.001, 0.01, 0.05, 0.1,0.5]

while(True):
    y, y_missing_count = get_least_missing_column(data)
#     y = 'sysjust8'
    new_data = data[get_filled_columns(data) + [y]].copy()

    # Convert categorical variables to dummy variables
    categorical_variables = list(set(cat_var) & set(get_filled_columns(new_data))
    new_data = pd.get_dummies(new_data, columns= categorical_variables, drop_firs

    # Ends here
    model_data = new_data[new_data[y].notnull()].copy()
    sample_index = random.sample(list(model_data.index), math.floor(hold_out_rati
    # Hold out data has the data based on teh hold out ratio
    hold_out_data = model_data.loc[sample_index].reset_index(drop = True)
    train_data = model_data.drop(sample_index, axis = 0).reset_index(drop = True)
    folds_data = kFold(train_data, k = folds)
    if y in num_var:
        print("num", y)
        best = -1
        for j in param1:
            for k in param2:
                error = []
                for i in range(folds):
                    training_data = folds_data[i]['Train_data']
                    validation_data = folds_data[i]['Val_data']
                    scaler = min_max_scaler(training_data)
                    training_data = scaler.transform(training_data)
                    validation_data = scaler.transform(validation_data)
                    ######## Insert the Model here and make changes accordingly
                    LR_Model = LinearRegression(method= j, lambda_value= k)
                    LR_Model.fit(training_data, y)
                    error.append(LR_Model.error(validation_data))
                if np.mean(error) < best or best == -1:
                    best = np.mean(error)
                    hold_out_data = scaler.transform(hold_out_data)
                    num_error[y] = LR_Model.error(hold_out_data)
                    best_model = LR_Model
        print("Error : ", num_error[y])
        for index, row in new_data[new_data[y].isna()].drop(y,axis = 1).iterrows(
            #print(index)
            row['Bias'] = 1.0
            data.at[index ,y] = best_model.predict_row(row)

    elif y in cat_var:
        print("cat", y)
```

```
        best = -1
        for j in param3:
            for k in param4:
                error = []
                for i in range(folds):
                    training_data = folds_data[i]['Train_data']
                    no_of_un_classes = len(training_data[y].unique())
#                     print("Classes: ", no_of_un_classes)

                    validation_data = folds_data[i]['Val_data']
                    scaler = min_max_scaler(training_data,y)
                    training_data = scaler.transform(training_data)
                    validation_data = scaler.transform(validation_data)
                    ######## Insert the Model here and make changes accordingly
                    if(no_of_un_classes == 2):
#                         print("Binary")
                        Log_Model = Logistic_Regression(X=training_data.drop(y,ax
                                                        no_of_epochs=j,learning_r
#                         print(np.any(training_data.drop(y,axis = 1).values))
#                         print(training_data.drop(y,axis = 1).isnull().values.an
                    else:
#                         print("Multi Class")
#                         Log_Model = MultiClassLogisticRegression(epochs = 100, l
                        Log_Model = MultiClass_Logistic_Regression(X=training_dat
                                                                   no_of_epochs=j
#                         print(training_data.drop(y,axis = 1).isnull().values.an
#                         Log_Model.fit()
#                         print(Log_Model.W)
#                         Log_Model.fit(training_data.drop(y,axis = 1).values,tra
#                         print(Log_Model.weights)
                    Log_Model.fit()
#                     print("Accuracy: ", Log_Model.accuracy(training_data.drop(y
                    X_valid= validation_data.drop(y,axis = 1)
                    Y_valid = validation_data[y]
                    error.append(Log_Model.error(X_valid,Y_valid))
                if np.mean(error) < best or best == -1:
                    best = np.mean(error)
                    hold_out_data = scaler.transform(hold_out_data)
                    hold_out_X_valid= hold_out_data.drop(y,axis = 1)
                    hold_out_y_valid = hold_out_data[y]
                    cat_error[y] = Log_Model.error(hold_out_X_valid,hold_out_y_va
                    best_model = Log_Model
        print("Error : ", cat_error[y])
        for index, row in new_data[new_data[y].isna()].drop(y,axis = 1).iterrows(
            #print(index)
            row=np.array(row)
            row=np.reshape(row,(1,len(row)))
            data.at[index ,y] = best_model.predict(row)
```

Plotting the errors

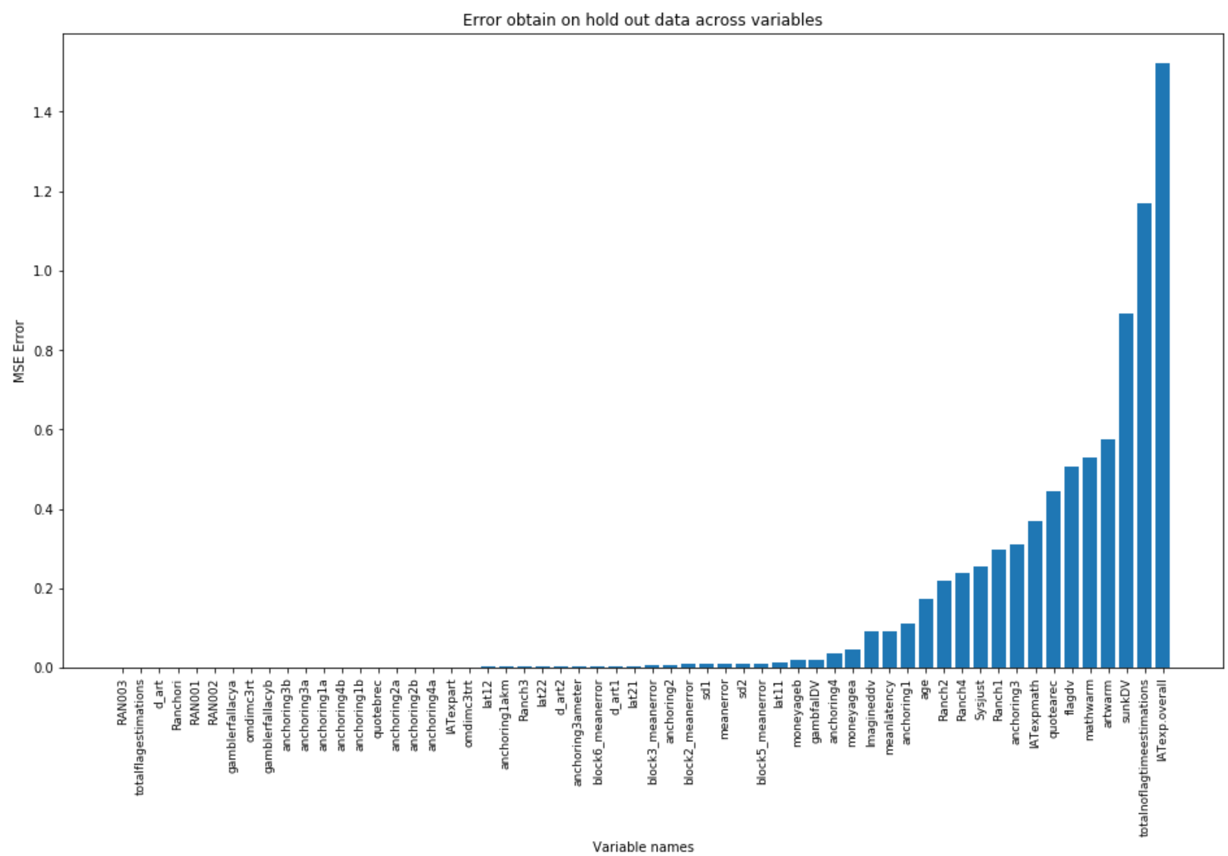
In [47]:
```
import operator

sorted_num_lr = sorted(num_error.items(), key=operator.itemgetter(1))
sorted_cat_lg = sorted(cat_error.items(), key=operator.itemgetter(1))
```

In [60]:
```
x = [x[0] for x in sorted_num_lr]
y = [x[1] for x in sorted_num_lr]
```

In [71]:
```
sum(y[:-1])
```

Out[71]:
```
8.017647677008966
```

In [52]:
```
plt.figure(figsize=(16,9))
plt.bar(x[:-1], y[:-1])
plt.xticks(fontsize = 9, rotation='90')
plt.xlabel('Variable names')
plt.ylabel('MSE Error')
plt.title('Error obtain on hold out data across variables')
plt.show()
```

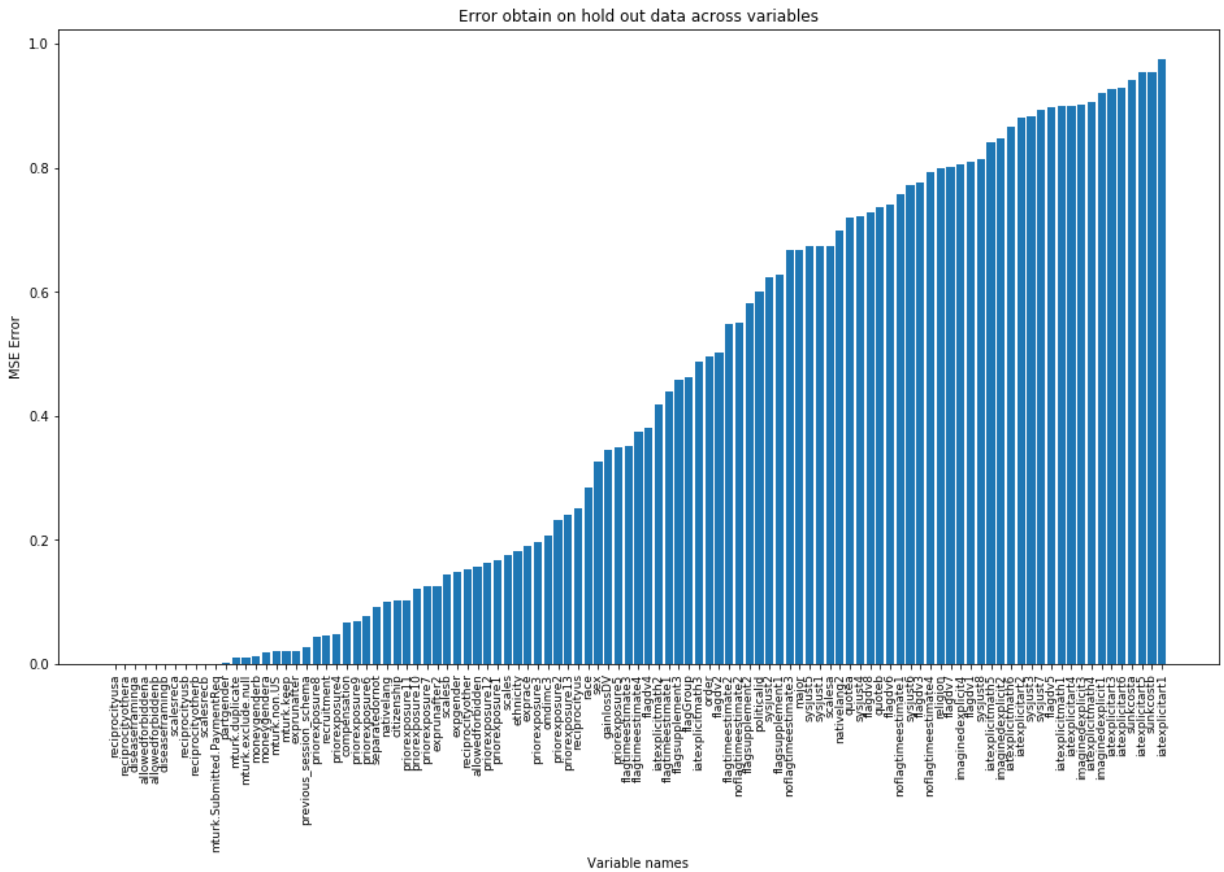


In [72]:
```
x = [x[0] for x in sorted_cat_lg]
y = [x[1] for x in sorted_cat_lg]
```

In [74]:
```
sum(y)
```

Out[74]:
```
44.229365231527886
```

In [54]:
```
plt.figure(figsize=(16,9))
plt.bar(x, y)
plt.xticks(fontsize = 9, rotation='90')
plt.xlabel('Variable names')
plt.ylabel('MSE Error')
plt.title('Error obtain on hold out data across variables')
plt.show()
```



Imputing using Neural Networks

In [113]:
```
class NeuralNetwork():

    def __init__(self, X = None , y = None, layers = [5, 2], learning_rate = 0.01
                epochs = 5, method = 'Linear', tol = 0.1, batch_size = 250):
        self.weights = None
        self.X = X
        self.y = y
        self.activationHidden = self.sigmoid
        self.method = method
        if self.method == 'Linear':
            self.activationOut = self.linear
            self.derivate_out = self.linear_der
            self.out_class = 'Linear'
        elif self.method == 'Classification' and len(np.unique(self.y)) == 2:
            self.out_class = 'Binary'
            self.activationOut = self.sigmoid
            self.derivate_out = self.sigmoid_der
        elif self.method == 'Classification' and len(np.unique(self.y)) > 2:
            self.out_class = 'MultiClass'
        self.layers = layers
            #self.activationOut = self.softmax
            #self.derivate_out = self.softmax_der
        self.derivate_rest = self.sigmoid_der
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.tol = tol
        self.batch_size = batch_size


    def weightsInitialisation(self):
        #Initialising a numpy array of dim(hiddenlayers, neurons) to store weight
        self.weights = []
        for i in range(len(self.layers)):
            temp = []
            for j in range(self.layers[i]):
                #first hidden layer
                if i == 0:
                    temp.append(np.random.normal(0,0.4, size = 1 + self.X.shape[1
                #rest hidden layers
                else:
                    temp.append(np.random.normal(0,0.4, size = 1 + self.layers[i-
            self.weights.append(temp)
        #Weights for the final output layer
        if self.out_class == 'MultiClass':
            self.outputLayerWeights =  np.random.normal(0,0.4, size = ( len(np.un
        else:
            self.outputLayerWeights =  np.random.normal(0,0.4, size = 1 + self.la

    def gradientInitialisation(self):
        self.gradient = []
        for i in range(len(self.layers)):
            temp = []
            for j in range(self.layers[i]):
                #first hidden layer
                if i == 0:
                    temp.append(np.zeros(1 + self.X.shape[1]))
```

```
                #rest hidden layers
                else:
                    temp.append(np.zeros(1 + self.layers[i-1]))
            self.gradient.append(temp)
        if self.out_class == 'MultiClass':
            self.gradientOutputLayer = np.zeros(shape = (len(np.unique(self.y)),
        else:
            self.gradientOutputLayer = [0] * len(self.outputLayerWeights)

    def sigmoid(self,x):
        if x < 0:
            return 1 - 1 / (1 + math.exp(x))
        else:
            return 1 / (1 + math.exp(-x))

    def linear(self,x):
        return x

    def sigmoid_der(self,x):
        return self.sigmoid(x) *(1 - self.sigmoid(x))

    def linear_der(self, x):
        return 1.0

    def softmax(self,x):
        shiftx = x - np.max(x)
        exps = np.exp(shiftx)
        return exps / np.sum(exps)

    def squareErrorLoss(self,x,y):
        return (self.feedForward(x) - y)**2

    def error(self, X, y):
        if self.out_class == 'Linear':
            pred= []
            for i in X:
                pred.append(self.feedForward(i))
            return mean([(a_i - b_i)**2 for a_i, b_i in zip(pred, y)])
        elif self.out_class == 'Binary':
            error = 0
            for i in range(len(X)):
                prob = self.feedForward(X[i])
                if (prob <0.5 and y[i] == 1) or (prob >=0.5 and y[i] == 0):
                    error = error + 1
            return error/X.shape[0]
        elif self.out_class == 'MultiClass':
            error = 0
            y = self.onehotencoding(y)
            for i in range(len(X)):
                prob = self.feedForward(X[i])
                class_pred = list(prob).index(max(prob))
                if class_pred != list(y[i]).index(1):
                    error = error + 1
            return error/X.shape[0]


    def predict(self,X):
```

```
        pred = []
        for i in X:
            pred.append(self.feedForward(i))
        return pred

    def predict_row(self,X):
        out = self.feedForward(X)
        if self.out_class == 'Linear':
            return out
        elif self.out_class == 'Binary':
            if out >= 0.5:
                return 1
            else:
                return 0
        elif self.out_class == 'MultiClass':
            return list(out).index(max(out))

    def loss(self, pred, actual):
        if self.method == 'Linear' or self.out_class == 'Binary':
            return 2.0 * (pred- actual)
        #elif self.out_class == 'Binary':
            #return
        elif self.out_class == 'MultiClass':
            p = np.dot(pred,actual)
            return (-1/math.log(p))

    def softmax_der(self, pred, actual, l):
        if actual[l] == 1:
            return pred[l]*(1 - pred[l])
        else:
            i = list(actual).index(1)
            return -1*pred[l]*pred[i]

    def onehotencoding(self, y):
        out = np.zeros((len(y),int(np.max(y)+1)))
        for i in range(len(y)):
            out[i][int(y[i])] = 1
        return out

    def feedForward(self, x):
        self.x = np.append(x, 1.0)
        self.out = []
        for i in range(len(self.layers) + 1):
            outputFromCurrLayer = []
            #For first Layer
            if i == 0:
                for j in range(self.layers[i]):
                    z = self.activationHidden(np.dot(self.weights[i][j],self.x))
                    outputFromCurrLayer.append(z)
                temp = outputFromCurrLayer.copy()
                self.out.append(temp)
                outputFromCurrLayer.append(1.0)
                outputFromPrevLayer = outputFromCurrLayer.copy()
            #Output Layer
            elif i == len(self.layers) and self.out_class == 'MultiClass':
                return self.softmax(np.matmul(self.outputLayerWeights, outputFrom
            elif i == len(self.layers):
```

```
            return self.activationOut(np.dot(self.outputLayerWeights,outputFr
        #Rest all Layers
        else:
            for j in range(self.layers[i]):
                z = self.activationHidden(np.dot(self.weights[i][j],outputFro
                outputFromCurrLayer.append(z)
            temp = outputFromCurrLayer.copy()
            self.out.append(temp)
            outputFromCurrLayer.append(1.0)
            outputFromPrevLayer = outputFromCurrLayer.copy()

def backProp(self, pred, actual):
    #Weight updation for Output Layer
    if self.out_class == 'Linear' or self.out_class == 'Binary':
        delta = []
        der_outter_layer = self.derivate_out(np.dot(np.append(self.out[len(se
        for i in range(len(self.outputLayerWeights)):
            if i == len(self.outputLayerWeights) - 1:
                self.gradientOutputLayer[i] = self.gradientOutputLayer[i] + (
            else :
                d = self.loss(pred, actual) * der_outter_layer * self.outputL
                self.gradientOutputLayer[i] = self.gradientOutputLayer[i] + (
                delta.append(d)
    elif self.out_class == 'MultiClass':
        delta = [0] * self.layers[-1]
        for l in range(len(self.outputLayerWeights)):
            der_outter_layer = self.softmax_der(pred,actual, l)
            for i in range(len(self.outputLayerWeights[l])):
                if i == len(self.outputLayerWeights[l]) - 1:
                    self.gradientOutputLayer[l][i] = self.gradientOutputLayer
                else:
                    d = self.loss(pred, actual) * der_outter_layer * self.out
                    delta[i] = delta[i] + d
                    self.gradientOutputLayer[l][i] = self.gradientOutputLayer

    #For all other Layers
    for l in reversed(range(len(self.layers))):
        delta_forward = delta.copy()
        delta = [0] * self.layers[l-1]
        #For the first layer
        if l == 0 :
            for j in range(self.layers[l]):
                der_layer = self.derivate_rest(np.dot(self.x , self.weights[l
                for i in range(len(self.weights[l][j])):
                    if i == len(self.weights[l][j]) - 1:
                        self.gradient[l][j][i] = self.gradient[l][j][i] +  (d
                    else :
                        self.gradient[l][j][i] = self.gradient[l][j][i] +   (
        #Rest all the Layers
        else :
            for j in range(self.layers[l]):
                der_layer = self.derivate_rest(np.dot(np.append(self.out[l -
                for i in range(len(self.weights[l][j])):
                    if i == len(self.weights[l][j]) - 1:
                        self.gradient[l][j][i] = self.gradient[l][j][i] +  (d
                    else :
                        d = delta_forward[j] * der_layer * self.weights[l][j]
```

```
                            delta[i] = delta[i] + d
                            self.gradient[l][j][i] = self.gradient[l][j][i] + (de

    def updateWeights(self, n):
        if self.out_class == 'Linear' and self.out_class == 'Binary':
            for i in range(len(self.outputLayerWeights)):
                self.outputLayerWeights[i] = self.outputLayerWeights[i] - (self.l
        elif self.out_class =='MultiClass':
            for l in range(len(self.outputLayerWeights)):
                for i in range(len(self.outputLayerWeights[l])):
                    self.outputLayerWeights[l][i] = self.outputLayerWeights[l][i]
        #For all other Layers
        for l in reversed(range(len(self.layers))):
            for j in range(self.layers[l]):
                for i in range(len(self.weights[l][j])):
                    self.weights[l][j][i] = self.weights[l][j][i] - (self.learnin


    def fit(self,X,y,X_val = None, Y_val = None):
        self.X = X
        self.y = y
        if self.out_class == 'MultiClass':
            y = self.onehotencoding(y)
        self.weightsInitialisation()
        self.gradientInitialisation()
        i = 0
        error_val_old = -1
        tol_count = 0
        while i < self.epochs:
            for j in range(len(X)):
                if j%self.batch_size ==0 and j != 0 or j == len(X) -1:
                    if j == len(X) -1:
                        self.updateWeights(j%self.batch_size)
                    else:
                        self.updateWeights(self.batch_size)
                    self.gradientInitialisation()
                    p = self.feedForward(X[j])
                    self.backProp(p,y[j])
                else:
                    p = self.feedForward(X[j])
                    self.backProp(p,y[j])
            #print(nn.weights)
#             if X_val is not None and Y_val is not None:
#                 error_curr_val = self.error(X_val, Y_val)
#                 print("Epoch : {} and MSE_Train : {} and MSE_Val : {}".format(i
#                 if abs(error_val_old -error_curr_val) < self.tol :
#                     tol_count = tol_count + 1
#                     error_val_old = error_curr_val
#                     if tol_count >1 :
#                         print("Stopping as validation error did not improve more
#                         break
#                 else:
#                     tol_count = 0
#                     error_val_old = error_curr_val
#             else:
#                 print("Epoch : {} and MSE : {}".format(i, self.error(X,y)))
            i = i+1
```

Pipeline for Neural Networks

In [93]:
```
# Initiation and parameters
num_error = {}
cat_error = {}
data = new_df.copy()
hold_out_ratio = 0.1
folds = 5
param1 = [[5,5], [3,3]]
param2 = [0.003, 0.05, 0.1]
param3 = [[5,5], [3,3]]
param4 = [0.003, 0.05, 0.1]

while(True):
    y, y_missing_count = get_least_missing_column(data)
#     y = 'sysjust8'
    new_data = data[get_filled_columns(data) + [y]].copy()

    # Convert categorical variables to dummy variables
    categorical_variables = list(set(cat_var) & set(get_filled_columns(new_data))
    new_data = pd.get_dummies(new_data, columns= categorical_variables, drop_firs

    # Ends here
    model_data = new_data[new_data[y].notnull()].copy()
    sample_index = random.sample(list(model_data.index), math.floor(hold_out_rati
    # Hold out data has the data based on teh hold out ratio
    hold_out_data = model_data.loc[sample_index].reset_index(drop = True)
    train_data = model_data.drop(sample_index, axis = 0).reset_index(drop = True)
    folds_data = kFold(train_data, k = folds)
    if y in num_var:
        print("num", y)
        best = -1
        for j in param1:
            for k in param2:
                error = []
                for i in range(folds):
                    training_data = folds_data[i]['Train_data']
                    validation_data = folds_data[i]['Val_data']
                    scaler = min_max_scaler(training_data)
                    training_data = scaler.transform(training_data)
                    validation_data = scaler.transform(validation_data)
                    ######## Insert the Model here and make changes accordingly
                    NN_Model = NeuralNetwork(X = training_data.drop(y, axis = 1).
                                             layers = j, learning_rate = k, metho
                    NN_Model.fit(X = training_data.drop(y, axis = 1).values, y =
                    error.append(NN_Model.error(validation_data.drop(y,axis = 1).
                if np.mean(error) < best or best == -1:
                    best = np.mean(error)
                    hold_out_data = scaler.transform(hold_out_data)
                    num_error[y] = NN_Model.error(hold_out_data.drop(y,axis = 1).
                    best_model = NN_Model
        print("Error : ", num_error[y])
        for index, row in new_data[new_data[y].isna()].drop(y,axis = 1).iterrows(
            data.at[index ,y] = best_model.predict_row(row)

    elif y in cat_var:
        print("cat", y)
        best = -1
```

```
        for j in param3:
            for k in param4:
                error = []
                for i in range(folds):
                    training_data = folds_data[i]['Train_data']
                    no_of_un_classes = len(training_data[y].unique())
                    validation_data = folds_data[i]['Val_data']
                    scaler = min_max_scaler(training_data,y)
                    training_data = scaler.transform(training_data)
                    validation_data = scaler.transform(validation_data)
                    ######## Insert the Model here and make changes accordingly
                    NN_Model = NeuralNetwork(X = training_data.drop(y, axis = 1).
                                             layers = j, learning_rate = k, metho
                    NN_Model.fit(X = training_data.drop(y, axis = 1).values, y =
                    error.append(NN_Model.error(validation_data.drop(y,axis = 1).
                if np.mean(error) < best or best == -1:
                    best = np.mean(error)
                    hold_out_data = scaler.transform(hold_out_data)
                    cat_error[y] = NN_Model.error(hold_out_data.drop(y,axis = 1).
                    best_model = NN_Model
        print("Error : ", cat_error[y])
        for index, row in new_data[new_data[y].isna()].drop(y,axis = 1).iterrows(
            data.at[index ,y] = best_model.predict_row(row)
```

Plotting the errors

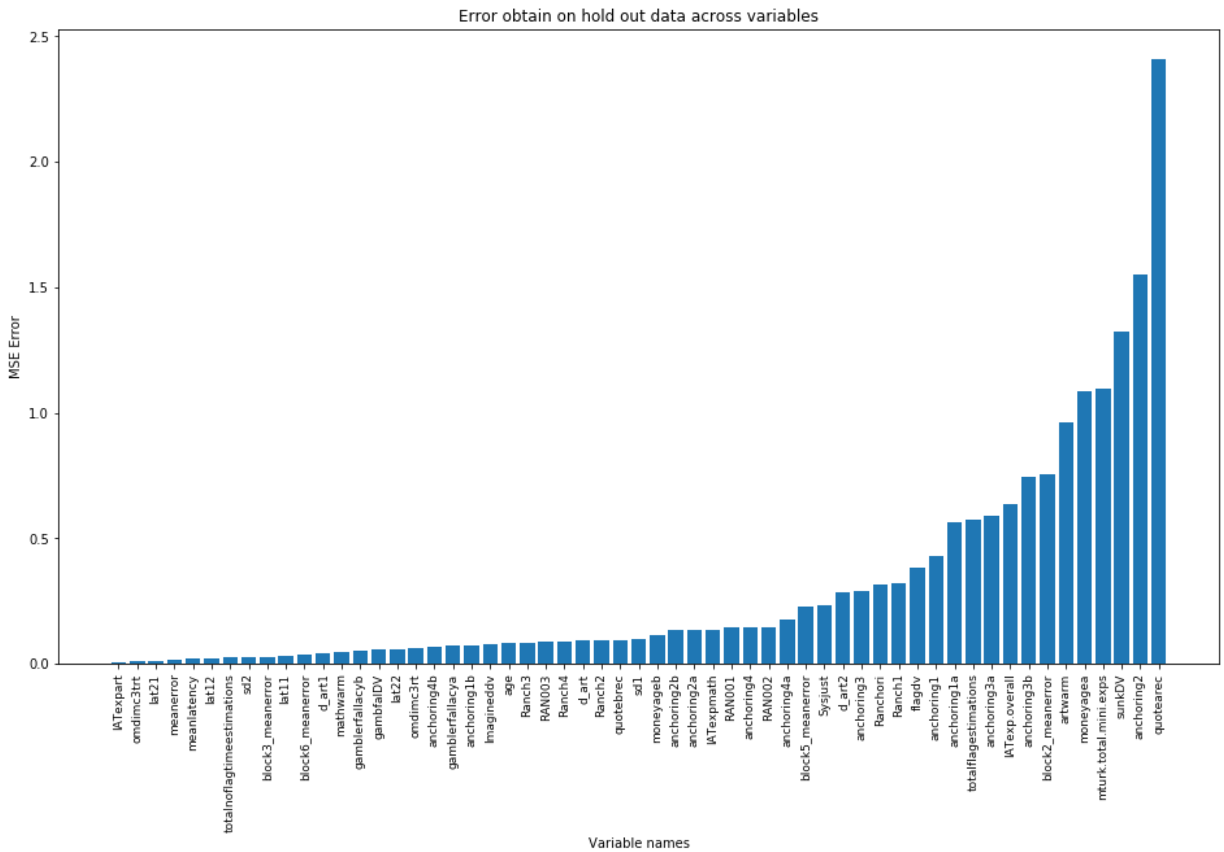
In [136]:

```
import operator

sorted_num_lr = sorted(num_error.items(), key=operator.itemgetter(1))
sorted_cat_lg = sorted(cat_error.items(), key=operator.itemgetter(1))
```

In [137]:

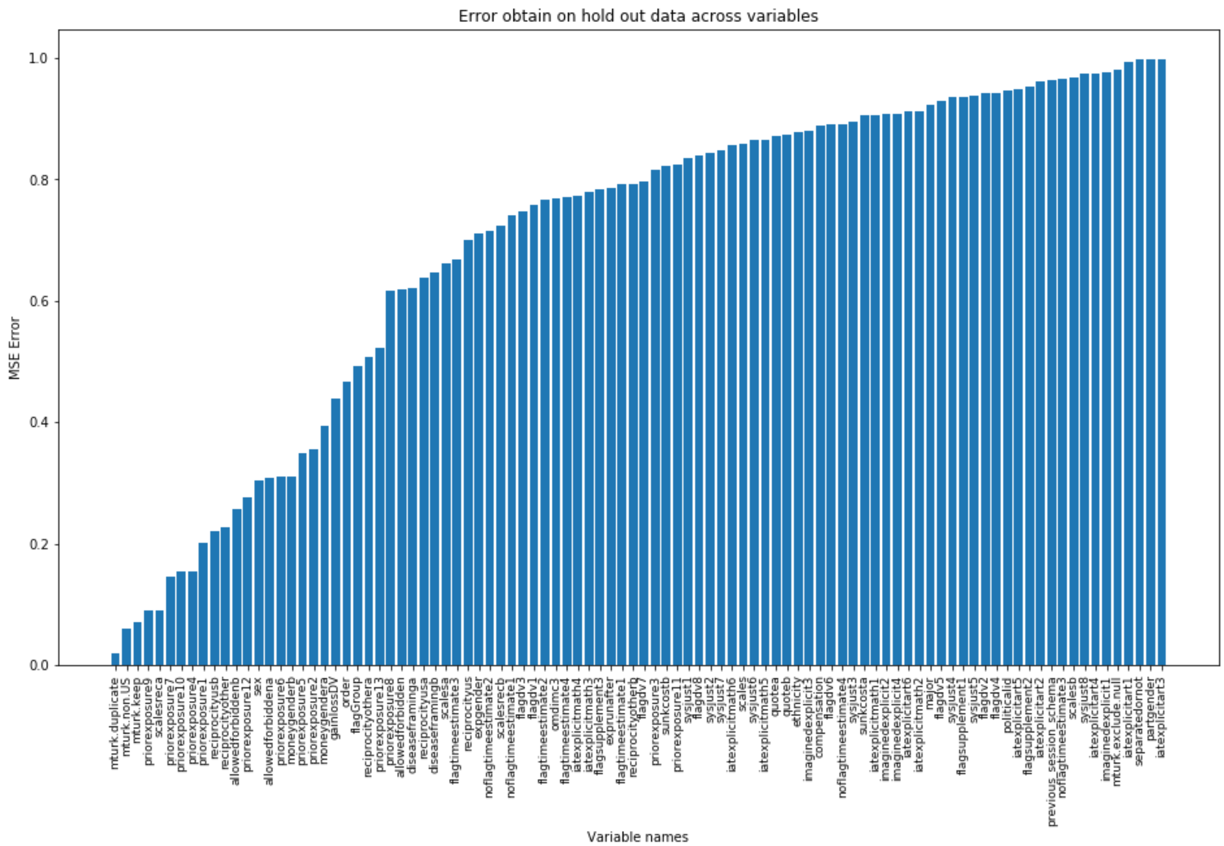
```
x = [x[0] for x in sorted_num_lr]
y = [x[1] for x in sorted_num_lr]
```

In [139]:
```
plt.figure(figsize=(16,9))
plt.bar(x, y)
plt.xticks(fontsize = 9, rotation='90')
plt.xlabel('Variable names')
plt.ylabel('MSE Error')
plt.title('Error obtain on hold out data across variables')
plt.show()
```



In [140]:
```
x = [x[0] for x in sorted_cat_lg]
y = [x[1] for x in sorted_cat_lg]
```

In [141]:
```
plt.figure(figsize=(16,9))
plt.bar(x, y)
plt.xticks(fontsize = 9, rotation='90')
plt.xlabel('Variable names')
plt.ylabel('MSE Error')
plt.title('Error obtain on hold out data across variables')
plt.show()
```



Imputation by PCA

In [87]:
```
def PCA(df, numerical, categorical):

    categorical_new =[]
    all_proportions = {}
    missing_flag = df.drop(columns = categorical).isna()

    for col in categorical:
        print(col)
        proportions = dict(df[col].value_counts(normalize=True))

        all_proportions[col] = proportions

        categories = df[col].unique()
        categories = list(categories)

        if (np.isnan(categories).any()):
            categories = [x for x in categories if ~np.isnan(x)]

        categorical_new+= categories

        for j in categories:
            if j is not np.nan:
                df[j] = 0

        for index, cat in enumerate(df[col]):

            if ((cat == np.nan) or np.isnan(cat)):
                for i in categories:
                    df[i].iloc[index] = np.nan
            else:
                df[cat].iloc[index] = 1

    missing_flag = df.drop(columns = categorical).isna()

    for old_col in categorical:

        categories = df[old_col].unique()

        if (np.isnan(categories).any()):
            categories = [x for x in categories if ~np.isnan(x)]

        for col in categories:
            df[col] = df[col].replace(np.nan, all_proportions[old_col][col])


    for col in numerical:
        df[col] = df[col].replace(np.nan,df[col].mean())#initially replacing with

    df = df.drop(columns =  categorical)
    missing_flag = df.isna()

    weights = np.zeros(df.shape)

    for i in range(len(missing_flag.values)):
        for j in range(len(missing_flag.values[0])):
            if missing_flag.values[i][j] == False:
```

```
                weights[i][j] = 1

    scaler = preprocessing.StandardScaler()
    scaled_df = scaler.fit_transform(df)
    U, sigma, V_transpose = np.linalg.svd(scaled_df)
    temp  = np.matmul( U[:,:df.shape[1]], np.diag(sigma))
    X_hat = np.matmul(temp,  V_transpose)
    X_new = np.multiply(weights,scaled_df) + np.multiply((1-weights), X_hat)
    tolerance = ((X_hat - X_new)**2).mean(axis=None)
    return (X_new,tolerance)
```

In [88]:
```
PCA_df = new_df.copy()
```

In [89]:
```
updated_df, tolerance = PCA(PCA_df.copy(), num_var, cat_var)
```