1.1

->Put raw csv in same path as ipynb for smooth running of code

This part of the code finds the number of houses in to whole dataset,
outputs ***clean_df*** csv for further processing,
marks out the time period of all malfunction meter

In [1]:

```
import numpy as np
import pandas as pd
from ipywidgets import FloatProgress
from IPython.display import display
import timeit
from datetime import timedelta
from pathlib import Path
# start_time = timeit.default_timer() * 1000
currentPath = str(Path().resolve())
# Put raw csv in same
path = currentPath + "./dataport-export_gas_oct2015-mar2016.csv"

# Read in raw data and find
df = pd.read_csv(path)
data_id = df['dataid'].nunique()
print("Number of houses =", data_id)
```

```
Number of houses = 157
```

In [2]:

```
# Sort data into order by id, if same id then by time
df.sort_values(["dataid", "localminute"], ascending=[True, True], inplace = True)
df.reset_index(drop = True, inplace = True)

# Cast localminute into proper time format for further processing
df.localminute = df.localminute.str.slice(0,19)
df.localminute = pd.to_datetime(df.localminute, infer_datetime_format = True, format =
"%Y/%m/%d %I:%M:%S %p")
print(df.localminute[0])
print("Length of df:", len(df))
```

```
2015-10-01 00:14:44
Length of df: 1584823
```

In [3]:

```
# Types of malfunction
# 1: data reported back when change in gas use is < 2 cubic foot
# 2: data reported back new meter_value is smaller than old meter_value
# 3: data reported back is >2, but time > 15s, threshold value for reporting malfunctio
n will be 7 cubic foot/hr
# (US household average daily usage = 168 cubic foot)
# 4: However, continuous report of same reading from same id over 12hrs,
# the first reading after 12hrs will be treated as good reading,
# as the gas company will probably want to know whether is a meter malfunctioning
# even though its reading did not change for a period of time

# Progress bar cuz I always feel it is not working
# f = FloatProgress(min=0, max=(len(df['dataid']) - 1))
# display(f)

# Create empty df and array for more efficient removal of item in df
malfunction = pd.DataFrame(columns = ["localminute", "dataid", "meter_value"])
bad_array = []
to_drop = []

prev_good = True
_id = None
_12hr = None
# Sort malfunction with time period label
# Assumption: the first data point is always correct as the 2nd pt is wrt to it, 3rd wr
t to 2nd....etc
# Append malfunction period and data to respective array
for row in df.itertuples():
    if(_id is None or _id != row.dataid):
        prev_good = True
        _id = row.dataid
    # for estimating time take to process actual data
    if (row.Index == 0):
        continue
    #1 and #2
    if (((row.meter_value <= df.meter_value[row.Index-1])
         and (row.dataid == df.dataid[row.Index-1]))
        or ((row.localminute != df.localminute[row.Index-1])
            and (row.dataid == df.dataid[row.Index-1])
            and ((row.meter_value - df.meter_value[row.Index-1]) < 2))):
        if(prev_good == True):
            bad_array.append([_id, row.localminute, row.localminute])
            to_drop.append(row.Index)
            #4, update 12hr pt
            _12hr = row.localminute
        else:
            #4
            if ((row.localminute - _12hr) >= pd.to_timedelta("12:00:00")):
                prev_good = True
                continue
            else:
                bad_array[-1][2] = row.localminute
                to_drop.append(row.Index)
        prev_good = False
    else:
        prev_good = True
        #3
        if(((row.meter_value - df.meter_value[row.Index-1]) > 2)
           and ((((row.localminute - df.localminute[row.Index-1]) / timedelta(hours = 1
```

```
)) * 7)
                > (row.meter_value - df.meter_value[row.Index - 1]))):
            bad_array[-1][2] = row.localminute
print("Length of bad_array:", len(bad_array))
print("Length of to_drop:", len(to_drop))
```

```
Length of bad_array: 169130
Length of to_drop: 1257748
```

In [4]:

```
#Remove malfunction data to produce clean df
df.drop(index = to_drop, inplace = True)
df.reset_index(drop = True, inplace = True)

# # Cast to int64
# malfunction['dataid'] = malfunction['dataid'].astype(np.int64)
# malfunction['meter_value'] = malfunction['meter_value'].astype(np.int64)

# # Merge 2 df to compare diff, if one of the label value is different,
# # value in _merge label will be different hence being able to differeniate the differ
ence
# df = pd.merge(df, malfunction, on=['localminute', 'dataid', 'meter_value'], how='oute
r', indicator=True)\
# .query("_merge != 'both'")\
# .drop(['_merge'], axis=1)\
# .reset_index(drop=True)

# Convert bad_array into df and transform it for shape to be correct
malfunction = pd.DataFrame(bad_array, columns = ['dataid', 'start_time', 'end_time'])
malfunction.T

# Save part1 result into csv for easier access in part2
df.to_csv('./clean_df.csv', index = False)
malfunction.to_csv('./malfunction.csv', index = False)

# pd.set_option('display.max_rows', 10000)

# elapsed = timeit.default_timer() * 1000 - start_time
# print("total: %ds" %(elapsed/1000)) if ((elapsed > 5000) == True) else print("total:
 %dms" %elapsed)
print("length of df:", len(df))
print("length of malfunction:", len(malfunction))
```

```
length of df: 327075
length of malfunction: 169130
```

1.2

In [5]:

```
import pandas as pd
import numpy as np
import timeit
import datetime as dt
import matplotlib.pyplot as plt
```

In [6]:

```
gas_data = pd.read_csv("clean_df.csv")
len(gas_data)
```

Out[6]:

```
327075
```

In [7]:

```
#preprocessing data for easier computation in the following
start_time = timeit.default_timer()
gas_data.localminute = gas_data.localminute.str.slice(0,19)
gas_data.localminute = pd.to_datetime(gas_data.localminute, infer_datetime_format = True,
                                      format = "%Y/%m/%d %I:%M:%S %p");
gas_data.localminute = gas_data.localminute.map(lambda x:x.replace(minute=0, second=0));
gas_data['meter_value']=gas_data['meter_value'].astype(float)
```

In [8]:

```
#gas_data=gas_data[gas_data['dataid']==35];
ind=0;
_hr=dt.timedelta(hours=1); #creating a constant timedelta object with value =1 hr for c
omparison
temp_gas_hr=pd.DataFrame(columns=gas_data.columns);
temp_gas_hr=gas_data;
id_list=gas_data['dataid'].unique();#creating an id list for itertaion since only compa
rison of reading within id
                                    #is meanningful

missing_lm=[];
missing_id=[];
missing_val=[];

for _id in id_list:
    #generate hourly readings according to dataid
    temp_gas_data=gas_data[gas_data['dataid']==_id];
    temp_gas_data.reset_index(drop=True,inplace=True);

    for  row in temp_gas_data.itertuples():
        if(row.Index==0):
            prev_row=pd.Series(data=[row.localminute,row.dataid,row.meter_value]
                               ,index=['localminute','dataid','meter_value']);
            #unable to predict datapoints before the first available datapoint
        else:
            time_diff=row.localminute-prev_row.localminute;
            #determine if there is any missing data before two consecutive available da
ta
            if(time_diff>_hr):
                time_diff=int(time_diff.total_seconds()/3600);
                #determine how many datapoints are missing
                for j in range (1,time_diff):
                    if ((row.meter_value-prev_row.meter_value)>4):
                        #if the difference between two available datepoint is too larg
e,
                        #this means that there is missing distinct datapoints in betwee
n.
                        acc_reading=float((row.meter_value-prev_row.meter_value)/time_d
iff);
                    else:
                        #if the difference is not large, the missing datapoint has valu
e
                        #equals to the previous datapoint
                        acc_reading=0;
                    time_change=dt.timedelta(hours=j);
                    new_time=prev_row.localminute+time_change;
                    missing_lm.append(new_time);
                    missing_id.append(_id);
                    missing_val.append(float(prev_row.meter_value+acc_reading*j));

        prev_row=pd.Series(data=[row.localminute,row.dataid,row.meter_value]
                               ,index=['localminute','dataid','meter_value']);
       #make a copy the current available data for comparison in next iteration
```

In [9]:

```
#concatenate the missing data with original data
missing_data={'localminute':missing_lm,'dataid':missing_id,'meter_value':missing_val};
missing_data=pd.DataFrame(missing_data,columns=gas_data.columns);
temp_gas_hr=pd.concat([gas_data,missing_data]);
```

In [10]:

```
temp_gas_hr=temp_gas_hr.sort_values(by=['dataid','localminute']);
temp_gas_hr.drop_duplicates(['localminute','dataid'],keep='last',inplace=True);
#for the same hour, if there is multiple readings, keep the highest value
temp_gas_hr['meter_value']=temp_gas_hr['meter_value'].astype(int)
print("total: %ds" %(timeit.default_timer() - start_time))
```

```
total: 164s
```

In [11]:

```
temp_gas_hr.to_csv('hourly_readings_final.csv',index=False);
```

Run this cell only if generating graphs

In [12]:

```
len(id_list)
```

Out[12]:

```
157
```

In [13]:

```
# gas_hr=temp_gas_hr;
# gas_hr=gas_hr[(gas_hr['localminute'].dt.year==2016)&(gas_hr['localminute'].dt.month==
1)]
# mon=2;
# for _id in id_list:
#     mon=2;
#     temp_id_hr=gas_hr[gas_hr['dataid']==_id];
#     while(len(temp_id_hr)==0):
#         temp_id_hr=temp_gas_hr[(temp_gas_hr['localminute'].dt.year==2016)&(temp_gas_h
r['localminute'].dt.month==mon)
#                                &(temp_gas_hr['dataid']==_id)];
#         mon=mon+1;
#     temp_id_hr=temp_id_hr.reset_index(drop=True);
#     t='Hourly reading of '+str(_id)+' of '+str(temp_id_hr.at[0,'localminute'].month)
+' in '+str(temp_id_hr.at[0,'localminute'].year);
#     fig=plt.figure(figsize=(15,15));
#     plt.plot(temp_id_hr.localminute,temp_id_hr.meter_value);
#     plt.xlabel('date hr/(hr)');
#     plt.ylabel('meter_value'),
#     plt.title(t,loc='center');
#     plt.grid();
#     t_fig=t+'.png';
#     fig.savefig(t_fig);
#     plt.show();
```

1.3

In [14]:

```
import pandas as pd
import numpy as np
import datetime as dt
from collections import namedtuple
```

In [15]:

```
# 1.3
hourly_reading = pd.read_csv('./hourly_readings_final.csv')

# get the list of dataids
idList = hourly_reading['dataid'].drop_duplicates(keep = 'first')
idList = idList.reset_index(drop = True)

len(idList)
```

Out[15]:

```
157
```

In [16]:

```
# split the gas meter readings based on data id
# create a dataframe with only meter readings
# the columns are all the different data ids
rd = pd.DataFrame(columns = idList)
for i in range(0, len(idList)):
    rd[idList[i]] = hourly_reading.loc[hourly_reading.dataid == idList[i]]['meter_valu
e'].reset_index(drop = True)

# compute the correlation matrix of the new dataframe
corr = rd.corr()

# remove the 1s in diagonal
# as the correlation between a column and itself is always 1
corr -= np.eye(corr.shape[0])
```

In [20]:

```
# create a new dataframe with random values inserted
top_corr = pd.DataFrame(np.random.randint(low = 0.0, high = 10.0, size = (5 * len(idLis
t), 3)), columns = ['HH1', 'HH2', 'corr']).reset_index(drop = True)

# set 'corr' column data type as float64
# since correlation value is a floating point between 0 and 1
top_corr['corr'] = top_corr['corr'].astype(float)

# loop through all data ids in the correlation matrix
# for each data id, find out the 5 data ids with the highest correlation values
# collate all results in the top_corr dataframe
count = 0
for i in corr.columns[:]:
    temp_corr = corr.nlargest(5, i)
    for j in range (0, 5):
        top_corr['HH1'].iloc[count] = i
        top_corr['HH2'].iloc[count] = temp_corr.index[j]
        top_corr['corr'].iloc[count] = temp_corr[i].iloc[j]
        count += 1

# export top_corr dataframe as csv file
top_corr.to_csv('top_correlated_households.csv', index = False)
```