2.1

1.Can you explain why you may want to forecast the gas consumption in the future?

The first reason why we want to predict is to detect gas leakage as soon as possible. By predicting the gas consumption in the future, when the actual consumption is far above (>50) than the prediction, it is very likely that there is gas leakage of the system. This is very important since the cost of gas leakage is very huge and we should detect this leakage as soon as possible and act on it. Hence, when there is a possible leakage, we should let the smart meter send a warning to both the gas company and the user to validate the situation.

The second reason why we want to predict is that we can adjust the gas supply at particular period. By prediction, the gas company can anticipate what is period that is likely to experience high demand. With this information, the gas company can adjust their supply at particular period to fulfill the demand and to keep less waste at period when there is less demand.

The third reason is that we can use the prediction to detect if the smart gas meter is malfunctioning. When the actual meter reading and the prediction behave very different, there is potential that the samrt meter is broken. Since malfunctionning of the meter might cause dispute in the final bill, the company should detect this and send in technician to fix it as soon as possible.

2.Who would find this information valuable?

Both the company and the customers will find it valuable

3.What can you do if you have a good forecasting model?

We can integrate with an IoT system for gas leakage detection and alarm and meter malfunction detection. The gas company can also use this prediction as a reference for gas generation.

2.2

In this section, we are building a prediction system using linear regression. To fit the data to the model, there are two processes that have been done.

Data Preproccessing:

Firstly, the time series has been converted to numbers. This is done by setting the earliest time point of the whole data set as 0 and increment of 1 meaning 1 hour later than the earliest time point(time_min). In this way, we are able to deal with time-series. When prediction occurs, we need to convert the time number back to a time point by adding the time number with time_min.

Secondly,in order to calculate consumption, we have initialize the first available reading of a particular id as the start point (consumption=0) and calculate consumption at particular time point by comparing the meter reading at that time with the reading at the start point. This is necessary as there are some id that has insufficient amount of data points and we want to replenish data of these id by comparing the id's consumption with other id's. If consumption is not extracted from original meter reading, it is not possible for us to have good replenishment for those data.

To train the model, since we have sufficient datas, the train to test ratio=9:1.

We only train for id with more than 10 data points. Since if the id has too little data, the prediction model would be meaningless.

In [ ]:

```
import pandas as pd
import numpy as np
import sklearn.linear_model as lm
from sklearn.svm import SVR
import sklearn.model_selection as model_sel
import datetime as dt
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
pd.options.mode.chained_assignment = None  # default='warn'
```

In [ ]:

```
gas_hr=pd.read_csv('./csv/hourly_readings_final.csv');
#id 6101 is dropped because it only has one data and there is no prediction can be done
 on one data point
id_list=gas_hr['dataid'].unique()
for _id in id_list:
    temp_data=gas_hr[gas_hr['dataid']==_id];
    if(len(temp_data)<10):
        gas_hr=gas_hr[gas_hr.dataid!=_id];

#Converting time varible to integer number
gas_hr.localminute = pd.to_datetime(gas_hr.localminute, infer_datetime_format = True,fo
rmat = "%Y/%m/%d %I:%M:%S %p");
min_time=min(gas_hr['localminute']);
gas_hr.localminute=gas_hr.localminute.map(lambda x: (x-min_time).total_seconds()/3600.0
);
```

In [ ]:

```
id_list=gas_hr['dataid'].unique()
len(id_list)
```

In [ ]:

```
normed_gas_hr=pd.DataFrame();
min_consump=[];

#Converting meter readings to consumption of all ids
for _id in id_list:
    gas_temp=gas_hr[gas_hr['dataid']==_id];
    min_consumption=min(gas_temp['meter_value']);
    gas_temp['meter_value']=gas_temp.meter_value.map(lambda x: x-min_consumption);
    min_consump.append(min_consumption);
    normed_gas_hr=normed_gas_hr.append(gas_temp);
```

This cell is meant to detect those id whose amount of data is deemed as insufficient. We have set the threshold point of amount of data to 2000 (2 months). We have extracted a list of ids and will subsititue more data points later.Since the subsitituded data is not the real data, it will only be used for training but not testing.

(let say we are subsitituting data for id A)

Firstly, we only use that that is within id A's data range. For example, if A only has data of Dec in 2015, we will only compare other id's data which is within Dec in 2015. The reason for this is that in our model, we assume strong relation between time and reading. Since time is assumed deterministic to actual reading, we can only consider data within the same period.

Secondly, the meteric used to determine if the data of another id can be used is consumption. For different id, the reading of the start point might not be the same, even though the consumption might be the same. Hence, by comparing readings directly, it is very likely that the comparison is not accurate.

Logic for subsititution of data:(let say we are subsitituting data for id A) 1. find out the time range of id A. 2. Extract data for all other ids within the same range. 3. loop through all other ids. If for id B, the available data has range that is narrower than id A, narrow id A's range to that range. Generate gas consumption data of the particular period. Divide B's consumption by A's and compare the results(div_result). A soft margin is set to allow for some discripency of consumption. Two point from id A and B are considered matched if div_result=1 or div_result = NaN(NaN is output of 0/0) and 1-soft margin<=div_result<=1+soft_margin. Compute number of points that are matched and see if this number exceeds x%(acceptance_lvl) of total number of data within the period. If yes, subsititute these data as id A's data. The smaller the soft margin and the higher the acceptance level, the less but more fitted data will be subed.

Data Pre-processing

In [ ]:

```
id_lack=[]
length=[]
thresh_length=2000;

#Determine what id need to be subed
for _id in id_list:
    gas_hr_id=gas_hr[gas_hr['dataid']==_id];
    length.append(len(gas_hr_id))
for index, item in enumerate(length):
    if((item<thresh_length) & (item>10)):
        id_lack.append(id_list[index]);
id_lack
```

In [ ]:

```
soft_margin=0.2;
acceptance_lvl=0.9;
#Create a varibale to store the subed data
normed_gas_hr_cp=pd.DataFrame(columns=['localminute','dataid','meter_value'])
for _id in id_lack:
    gas_lack_hr=normed_gas_hr[normed_gas_hr['dataid']==_id];
    time_min=min(gas_lack_hr['localminute']);
    time_max=max(gas_lack_hr['localminute']);

    #Extract id's who has data that is within time range we are looking for
    gas_compensate=normed_gas_hr[(normed_gas_hr['localminute']>=time_min) & (normed_gas
_hr['localminute']<=time_max)
                         &(normed_gas_hr['dataid']!=_id)];
    id_com=gas_compensate['dataid'].unique();

    #Loop through all candidate ids, searching for subsititution
    for c_id in id_com:
        #Find out the range of data for comparison and compute consumption based on rea
dings
        gas_cid=pd.DataFrame();
        gas_cid=gas_compensate[gas_compensate['dataid']==c_id];
        min_consumption=min(gas_cid['meter_value']);
        gas_cid['meter_value']=gas_cid.meter_value.map(lambda x: x-min_consumption);
        gas_cid.reset_index(drop=True,inplace=True);

        #If id B has a smaller range than id A
        #Extract data from id A within the same range and compute for consumption
        time_min_c=min(gas_cid['localminute']);
        time_max_c=max(gas_cid['localminute']);
        gas_lack_c=pd.DataFrame();
        if((time_min_c!=time_min) | (time_max_c!=time_max)):
            gas_lack_c=gas_lack_hr[(gas_lack_hr['localminute']>=time_min_c) & (gas_lack
_hr['localminute']<=time_max_c)];
            min_consumption=min(gas_lack_c['meter_value']);
            gas_lack_c['meter_value']=gas_lack_c.meter_value.map(lambda x: x-min_consum
ption);
        else:
            gas_lack_c=gas_lack_hr;
        gas_lack_c.reset_index(drop=True,inplace=True);

        #Compute difference between two data set by division
        diff_percent=gas_cid['meter_value']/gas_lack_c['meter_value'];
        #Find out the amount of matched data points
        equal_num=len(diff_percent[(diff_percent<=(1+soft_margin))&
                                   (diff_percent>=(1-soft_margin))])+len(diff_percent[n
p.isnan(diff_percent)]);

        #From the number of matched data points, determine if the data can be used as s
ubsitition
        if(equal_num>=acceptance_lvl*len(gas_cid)):
            gas_cid.dataid=gas_cid.dataid.replace(c_id,_id);
            if(time_min_c!=time_min):
                temp_reading=gas_lack_hr[gas_lack_hr['localminute']==time_min_c]['meter
_value'].values;
                extra_reading=temp_reading[0];
                gas_cid['meter_value']=gas_cid.meter_value.map(lambda x: x+extra_readin
g);
            normed_gas_hr_cp=normed_gas_hr_cp.append(gas_cid);
```

In [ ]:

```
#Plots to see what has been subed for the lacking ids

# for _id in id_lack:
#     temp=normed_gas_hr_cp[normed_gas_hr_cp['dataid']==_id]
#     temp.reset_index(drop=True,inplace=True)
#     fig=plt.figure(figsize=(5,5));
#     plt.plot(temp.localminute,temp.meter_value,'r');
#     plt.xlabel('date hr/(hr)');
#     plt.ylabel('meter_value'),
#     plt.grid();
#     plt.show();
```

Model Training

LR

In [ ]:

```
P_array=[];
score=[];
decre_index=0;
id_list1=[35]
all_result=pd.DataFrame(columns=['localminute','dataid','actual_reading','predicted_rea
ding']);

for index,item in enumerate(id_list):
    #Extract attributes and wanted values
    X_temp=normed_gas_hr[normed_gas_hr['dataid']==item][['localminute']];
    Y_temp=normed_gas_hr[normed_gas_hr['dataid']==item][['meter_value']];
    #Split train and test case
    (X_train, X_test, Y_train, Y_test)=model_sel.train_test_split(X_temp,Y_temp,test_si
ze=0.1);

    #For lacking ids add in more training data
    if(item in id_lack):
        X_train=X_train.append(normed_gas_hr_cp[normed_gas_hr_cp['dataid']==item][['loc
alminute']]);
        Y_train=Y_train.append(normed_gas_hr_cp[normed_gas_hr_cp['dataid']==item][['met
er_value']]);

    #only train for model has more than 10 data points, since the model will be more or
 less meaningless with
    #very little amout of data
    if(len(X_temp)<10):
        P_array.append('not enough data');
        score.append('not enough data');
    else:
        #Training using LR model
        LR=lm.LinearRegression();
        predictor=LR.fit(X_train,Y_train);
        #Stor the trained predictor and the score of the prefictor for future use
        P_array.append(predictor)
        score.append(P_array[index].score(X_test,Y_test));
        #Predict for test data for comparison
        predicted=P_array[index].predict(X_test);

        #Convert back the time number to time point
        #Convert back consumption to meter readings
        X_test=X_test.localminute.map(lambda x: dt.timedelta(hours=x)+min_time);
        Y_test=Y_test.meter_value.map(lambda x:x+min_consump[index]);
        predicted=np.add(predicted,min_consump[index]);
        result=pd.DataFrame(columns=['localminute','dataid','actual_reading','predicted
_reading']);
        result['localminute']=X_test;
        result['actual_reading']=Y_test;
        result['predicted_reading']=predicted;
        result['dataid']=item;
        result=result.sort_values(by='localminute');
        #Store the predicted result for plots in the next section
        all_result = all_result.append(result)

        #Plot of prediction vs actual reading against time
        fig=plt.figure(figsize=(10,10));
        t='Prediction & Actual reading vs time of '+str(item)
        plt.plot(result.localminute,result.actual_reading,'r',label='Actual');
        plt.plot(result.localminute,result.predicted_reading,'b',label='Predicted');
        plt.legend()
        plt.xlabel('date hr/(hr)');
        plt.ylabel('consumption');
        plt.title(t,loc='center');
        plt.grid();
        plt.show();
```

In [ ]:

```
#Scatter plot of prediction vs actual reading
valid_id=all_result['dataid'].unique()
for _id in valid_id:
    temp=all_result[all_result['dataid']==_id];
    fig=plt.figure(figsize=(10,10));
    t='Prediction vs Actual of '+str(_id)+' with fit line(y=x)'
    plt.scatter(temp.predicted_reading.tolist(),temp.actual_reading.tolist());
    plt.plot(temp.predicted_reading,temp.predicted_reading,'r',label='y=x')
    plt.legend();
    plt.xlabel('Predicted reading');
    plt.ylabel('Actual reading');
    plt.title(t,loc='center');
    plt.grid();
    plt.show();
```

SVR

In [ ]:

```
P_array=[];
score=[];
decre_index=0;
id_list1=[35]
all_result=pd.DataFrame(columns=['localminute','dataid','actual_reading','predicted_rea
ding']);

for index,item in enumerate(id_list):
    #Extract attributes and wanted values
    X_temp=normed_gas_hr[normed_gas_hr['dataid']==item][['localminute']];
    Y_temp=normed_gas_hr[normed_gas_hr['dataid']==item][['meter_value']];
    #Split train and test case
    (X_train, X_test, Y_train, Y_test)=model_sel.train_test_split(X_temp,Y_temp,test_si
ze=0.1);

    #For lacking ids add in more training data
    if(item in id_lack):
        X_train=X_train.append(normed_gas_hr_cp[normed_gas_hr_cp['dataid']==item][['loc
alminute']]);
        Y_train=Y_train.append(normed_gas_hr_cp[normed_gas_hr_cp['dataid']==item][['met
er_value']]);

    # Reshape Y
    Y_train = Y_train.meter_value.ravel()

    #only train for model has more than 10 data points, since the model will be more or
 less meaningless with
    #very little amout of data
    if(len(X_temp)<10):
        P_array.append('not enough data');
        score.append('not enough data');
    else:
        #Training using SVR model
        LR = SVR(C = 3000, epsilon = 20, gamma = 1e-5)
        predictor=LR.fit(X_train,Y_train);
        #Store the trained predictor and the score of the prefictor for future use
        P_array.append(predictor)
        score.append(P_array[index].score(X_test,Y_test));
        #Predict for test data for comparison
        predicted=P_array[index].predict(X_test);

        #Convert back the time number to time point
        #Convert back consumption to meter readings
        X_test=X_test.localminute.map(lambda x: dt.timedelta(hours=x)+min_time);
        Y_test=Y_test.meter_value.map(lambda x:x+min_consump[index]);
        predicted=np.add(predicted,min_consump[index]);
        result=pd.DataFrame(columns=['localminute','dataid','actual_reading','predicted
_reading']);
        result['localminute']=X_test;
        result['actual_reading']=Y_test;
        result['predicted_reading']=predicted;
        result['dataid']=item;
        result=result.sort_values(by='localminute');
        #Store the predicted result for plots in the next section
        all_result = all_result.append(result)

        #Plot of prediction vs actual reading against time
        fig=plt.figure(figsize=(10,10));
        t='Prediction & Actual reading vs time of '+str(item)
        plt.plot(result.localminute,result.actual_reading,'r',label='Actual');
        plt.plot(result.localminute,result.predicted_reading,'b',label='Predicted');
        plt.legend()
        plt.xlabel('date hr/(hr)');
        plt.ylabel('consumption');
        plt.title(t,loc='center');
        plt.grid();
        plt.show();
```

In [ ]:

```
#Scatter plot of prediction vs actual reading
valid_id=all_result['dataid'].unique()
for _id in valid_id:
    temp=all_result[all_result['dataid']==_id];
    fig=plt.figure(figsize=(10,10));
    t='Prediction vs Actual of '+str(_id)+' with fit line(y=x)'
    plt.scatter(temp.predicted_reading.tolist(),temp.actual_reading.tolist());
    plt.plot(temp.predicted_reading,temp.predicted_reading,'r',label='y=x')
    plt.legend();
    plt.xlabel('Predicted reading');
    plt.ylabel('Actual reading');
    plt.title(t,loc='center');
    plt.grid();
    plt.show();
```

In [ ]:

```
# Tuning parameters
parameters = [{'kernel': ['rbf'],'gamma': [1e-4, 1e-5], 'C': [2000, 3000], 'epsilon' :
[50, 100]}]
```

In [ ]:

```
# 5-fold cross validation

clf = GridSearchCV(SVR(), parameters, cv=5)
clf.fit(X_train, Y_train)

print("Best parameters set found on development set:")
print()
print(clf.best_params_)
print()
print("Grid scores on training set:")
print()
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r"
          % (mean, std * 2, params))
print()
```

Conclusion

As seen from the plots, the performance of the LR predictor is not very good. That is mainly because many id's consumption pattern is not linear in nature, but rather looks like a polynomial. Hence by applying LR to predict, the outcome is not very satisfactory.

In comparison, the performance of SVR is much better than LR as it is able to separate linearly non separable data by doing the kernel trick at higher dimensions, as well as controlling how close the fit is the model to the data, resulting in a closely represented model of the data. Parameters of SVR is selected based with 5-fold cross validation, and visual inspection of the resultant graph. As it is observed that the validation will try to fit to the graph very closely, we decide to stop at a certain which we feel that tuning the parameters more will only result in overfitting. Else it will only result in a very large C value, very large epsilon value and a small gamma value.

In conclusion, SVR is more robust at predicting data compared to LR if the data is non linear.

3 Clustering and analysis of residential customers' gas consumption behavioral using aggregated smart meter data

Following topics will be discussied: introduction, feature extraction, analysis on number of clusters chosen, k-means clustering implementation, user behavior analysis based on clustering, applying price discrimination on different clusters and conclusion.

3.1 Introduction

The wide spread of smart metering roll out enables a better understanding of the consumer behavior and tailoring demand response to achieve cost-efficient energy savings. In the residential sector smart metering allows detailed readings of the power consumption in the form of large volumes time series that encodes relevant information to be found. In our project, we would like to analyze the gas consumption data further by applying clustering method to measure the similarity between the households, and group them based on energy usage. The goal in clustering time-series data is to understand user behavior by organizing the data into homogeneous groups, maximizing the similarity and dissimilarity within and between groups, respectively.

Clustering the most important attributes (more in section 3.2 feature extraction) of customers is a very common method for better understanding the different residential energy behaviours that exist and has many applications (Stephen,2015). Potential application areas for work using this form of cluster analysis as more smart data becomes routinely available include

1. Application of cluster results for segment-specific rate design: the economic benefits of time-variable natural gas rates consist in the potential for utilities to improve price discrimination and to facilitate the reduction of peak loads. A complete design of gas scheme would encompass number of different time zones and starting time of each period. All these could be done through clustering analysis. In addition, by influencing the demand side, segment-specific natural gas rates can improve the energy system's efficiency.
2. Application of cluster results for leak detection activities based on detecting pattern changes (deviation from cluster centroids/ distributions); data-driven models of demand could also help identify atypical customers or unusual changes in consumption.
3. Application of cluster results for filling missing data for audits/ regulatory purposes using cluster centroids. Typical usage perhaps allows volumetric usage and flow profiles to be estimated for unmetered customers.

3.2 Feature extraction

Clustering is a collection of same group similar or data objects or in other groups. Its also finding the dissimilar to the data objects in other groups. In cluster analysis, the main objective is to find similarities between data objects with the help of specific characteristics found in the data and grouping these similar data objects into clusters. Large quantities of information about how customers use their energy is then becoming available through the uptake of smart meters. In our project, the following features and their respective standard deviation will be examined in the following clustering discussion.

Features	Discription
Total consumption	Total consumed power during period of the study
Morning consumtption	The consumption observed from 06:00 - 09:00
Noon consumtption	The consumption observed from 11:00 - 14:00
Night consumtption	The consumption observed from 17:00 - 20:00

In [ ]:

```
import numpy as np
import pandas as pd
from ipywidgets import FloatProgress
from IPython.display import display
from pathlib import Path
import timeit
import datetime
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import pylab as pl
from sklearn.decomposition import PCA
from IPython.display import HTML
from sklearn.datasets import load_iris
from sklearn import preprocessing
from sklearn.metrics import silhouette_samples, silhouette_score
import matplotlib.cm as cm
from __future__ import print_function
```

Generate a csv file from hourly_readings_final.csv for easier computation in the later sections

In [ ]:

```
# import result obtained from 1.2
hourly_reading = pd.read_csv('./csv/hourly_readings_final.csv')

# split the column 'localminute' into 'day' and 'time'
new_time = hourly_reading['localminute'].str.split(" ", n = 1, expand = True)
hourly_reading['day'] = new_time[0]
hourly_reading['time'] = new_time[1]
hourly_reading.drop(columns = ['localminute'], inplace = True)

# add the 'consumption' column which shows the hourly consumption by each household
# the function diff() computes the difference between the current data point and the pr
evious data point
# therefore the data points at the boundary (data id changes) are wrong and will be cha
nged to 0
hourly_reading['consumption'] = hourly_reading['meter_value'].diff()
mask = (hourly_reading.dataid != hourly_reading.dataid.shift(1))
hourly_reading.loc[mask, 'consumption'] = 0
hourly_reading.to_csv(path+"/hourly_readings_with_consumption.csv", index = False)
```

In [ ]:

```
# headers include all features needed for later discussion
df = pd.read_csv(path+"/hourly_readings_with_consumption.csv")

headers = ["dataid", "totalConsumption", "morningTotal","morningMean", "morningSTD", "n
oonTotal","noonMean",
          "noonSTD", "nightTotal","nightMean", "nightSTD","elseConsumption"]
currentID = df.iloc[0]['dataid']
baseTime = df.iloc[0]['time']
featuresList = []
morningConsumption = []
noonConsumption = []
nightConsumption = []
totalConsumption = 0
elseConsumption = []
currentMorning = 0
currentNoon = 0
currentNight = 0
currentElse = 0
prevDay = df.iloc[0]['day']
```

In [ ]:

```
# Feature Extraction
for row in df.itertuples():
    currentTime = row.time
    hrs_diff = int(currentTime.split(":")[0][:2])
    currentDay = row.day

    # For different days, we calculate their morning, noon and night consumption
    if (currentDay != prevDay):
        morningConsumption.append(currentMorning)
        noonConsumption.append(currentNoon)
        nightConsumption.append(currentNight)
        elseConsumption.append(currentElse)
        prevDay = currentDay
        currentMorning = 0
        currentNoon = 0
        currentNight = 0
        currentElse = 0

    # For different ID, we append respective features to the list
    if (currentID != row.dataid):
        features = []
        morningTotal = np.sum(morningConsumption)
        morningMean = np.mean(morningConsumption)
        morningSTD = np.std(morningConsumption)

        noonTotal = np.sum(noonConsumption)
        noonMean = np.mean(noonConsumption)
        noonSTD = np.std(noonConsumption)

        nightTotal = np.sum(nightConsumption)
        nightMean = np.mean(nightConsumption)
        nightSTD = np.std(nightConsumption)
        elseConsumptionTotal = np.sum(elseConsumption)

        features.append(currentID)
        features.append(totalConsumption)
        features.append(morningTotal)
        features.append(morningMean)
        features.append(morningSTD)
        features.append(noonTotal)
        features.append(noonMean)
        features.append(noonSTD)
        features.append(nightTotal)
        features.append(nightMean)
        features.append(nightSTD)

        features.append(elseConsumptionTotal)
        featuresList.append(features)

        elseConsumption =[]
        morningConsumption = []
        noonConsumption = []
        nightConsumption = []
        totalConsumption = 0
        currentID = row.dataid

    # Feature calculation
    totalConsumption = totalConsumption + int(row.consumption)
    if (hrs_diff >= 6 and hrs_diff <= 9):
            currentMorning = currentMorning + int(row.consumption)
        elif (hrs_diff >= 11 and hrs_diff <= 14):
            currentNoon = currentNoon + int(row.consumption)
        elif (hrs_diff >= 17 and hrs_diff <= 20):
            currentNight = currentNight + int(row.consumption)
        else:
            currentElse = currentElse + int(row.consumption)

morningConsumption.append(currentMorning)
noonConsumption.append(currentNoon)
nightConsumption.append(currentNight)
morningMean = np.mean(morningConsumption)
morningSTD = np.std(morningConsumption)
noonMean = np.mean(noonConsumption)
noonSTD = np.std(noonConsumption)
nightMean = np.mean(nightConsumption)
nightSTD = np.std(nightConsumption)
features = []

# We append the last ID features
features.append(currentID)
features.append(totalConsumption)
features.append(morningTotal)
features.append(morningMean)
features.append(morningSTD)
features.append(noonTotal)
features.append(noonMean)
features.append(noonSTD)
features.append(nightTotal)
features.append(nightMean)
features.append(nightSTD)
features.append(elseConsumptionTotal)
featuresList.append(features)

featureFrame = pd.DataFrame(featuresList)
featureFrame.to_csv(path + '/reading_features.csv', header=headers, index=None)
```

The feature values are normalized to values between 0 and 1 in order to stabilize the algorithm to differences in scale across features.

In [ ]:

```
# Normalisation of feature values
df = pd.read_csv(path+"/reading_features.csv")
featureNames = ["totalConsumption", "morningTotal", "noonTotal", "nightTotal"]
# featureNames = ["totalConsumption", "morningMean", "noonMean", "nightMean"]

# Normalise the data for range o to 1
x = df[featureNames].values.astype(float)
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
x_scaled = min_max_scaler.fit_transform(x)
df_normalized = pd.DataFrame(x_scaled)
```

3.3 Analysis on number of clusters chosen

In order to determine number of clusters we should have, silhouette coefficient is used to quantify the quality of clustering achieved. We will select the number of clusters that maximizes the silhouette coefficient. The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette coefficient is calculated as followed and the calculation is implemented in the code below. For each point p, first find the average distance between p and all other points in the same cluster (this is a measure of cohesion, call it A). Then find the average distance between p and all points in the nearest cluster (this is a measure of separation from the closest other cluster, call it B). The silhouette coefficient for p is defined as the difference between B and A divided by the greater of the two (max (A, B)). Figure above provides a silhouette plot for k-means applied with different cluster numbers ranging from 2 to 6. From all the plots below, 3 clusters have a highest average silhouette score which is the optimum solution in our case and this will be used for further discussion.

In [ ]:

```
# We test on different cluster numbers: 2, 3, 4, 5 and 6
range_n_clusters = [2, 3, 4, 5, 6]
X= df_normalized.values

for n_clusters in range_n_clusters:
    # Create a subplot with 1 row and 2 columns
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)

    # The 1st subplot is the silhouette plot
    # In our example, the silhouette coefficient can range from -0.1. The higher, the b
etter.
    ax1.set_xlim([-0.1, 1])
    # The (n_clusters+1)*10 is for inserting blank space between silhouette
    # plots of individual clusters, to demarcate them clearly.
    ax1.set_ylim([0, len(X) + (n_clusters + 1) * 10])

    # Initialize the clusterer with n_clusters value and a random generator
    # seed of 10 for reproducibility.
    clusterer = KMeans(n_clusters=n_clusters, random_state=10)
    cluster_labels = clusterer.fit_predict(X)

    # The silhouette_score gives the average value for all the samples.
    # This gives a perspective into the density and separation of the formed
    # clusters
    silhouette_avg = silhouette_score(X, cluster_labels)
    print("For n_clusters =", n_clusters,
          "The average silhouette_score is :", silhouette_avg)

    # Compute the silhouette scores for each sample
    sample_silhouette_values = silhouette_samples(X, cluster_labels)

    y_lower = 10
    for i in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to
        # cluster i, and sort them
        ith_cluster_silhouette_values = \
            sample_silhouette_values[cluster_labels == i]

        ith_cluster_silhouette_values.sort()

        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i

        color = cm.nipy_spectral(float(i) / n_clusters)
        ax1.fill_betweenx(np.arange(y_lower, y_upper),
                          0, ith_cluster_silhouette_values,
                          facecolor=color, edgecolor=color, alpha=0.7)

        # Label the silhouette plots with their cluster numbers at the middle
        ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))

        # Compute the new y_lower for next plot
        y_lower = y_upper + 10  # 10 for the 0 samples

    ax1.set_title("The silhouette plot for the various clusters.")
    ax1.set_xlabel("The silhouette coefficient values")
    ax1.set_ylabel("Cluster label")
    # The vertical line for average silhouette score of all the values
    ax1.axvline(x=silhouette_avg, color="red", linestyle="--")

    ax1.set_yticks([])  # Clear the yaxis labels / ticks
    ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

    # 2nd Plot showing the actual clusters formed
    colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
    ax2.scatter(X[:, 0], X[:, 1], marker='.', s=30, lw=0, alpha=0.7,
                c=colors, edgecolor='k')

    # Labeling the clusters
    centers = clusterer.cluster_centers_
    # Draw white circles at cluster centers
    ax2.scatter(centers[:, 0], centers[:, 1], marker='o',
                c="white", alpha=1, s=200, edgecolor='k')

    for i, c in enumerate(centers):
        ax2.scatter(c[0], c[1], marker='$%d$' % i, alpha=1,
                    s=50, edgecolor='k')

    ax2.set_title("The visualization of the clustered data.")
    ax2.set_xlabel("Feature space for the combined 1st feature")
    ax2.set_ylabel("Feature space for the combined 2nd feature")

    plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "
                  "with n_clusters = %d" % n_clusters),
                 fontsize=14, fontweight='bold')
plt.show()
```

3.4 K-means clustering implementation

K-medoids clustering selects the most centrally located data points within clusters as cluster centers called medoids and returns the number of clustering as required.

In [ ]:

```
kmeans = KMeans(n_clusters=3).fit(df_normalized)
df.insert(loc=0, column="classification", value=kmeans.labels_)
df.to_csv(path + '/reading_features_classification.csv', index=None)
```

In [ ]:

```
# 2D Visualization of Data plots
fig, (ax3, ax4,ax5) = plt.subplots(1, 3)
fig.set_size_inches(22, 7)

df_array = df.values
ax3.scatter(df_array[:,2],df_array[:,4], c=kmeans.labels_, cmap='rainbow')
ax3.set_title("The visualization of the clustered data.")
ax3.set_xlabel('Total Consumption')
ax3.set_ylabel('Morning Average Consumption')

ax4.scatter(df_array[:,2],df_array[:,7], c=kmeans.labels_, cmap='rainbow')
ax4.set_title("The visualization of the clustered data.")
ax4.set_xlabel('Total Consumption')
ax4.set_ylabel('Noon Average Consumption')

ax5.scatter(df_array[:,2],df_array[:,10], c=kmeans.labels_, cmap='rainbow')
ax5.set_title("The visualization of the clustered data.")
ax5.set_xlabel('Total Consumption')
ax5.set_ylabel('Night Average Consumption')
```

In [ ]:

```
# Combine all 4 feature into 2 feature sets
pca = PCA(n_components=2).fit(df[featureNames])
pca_2d = pca.transform(df[featureNames])
for i in range(0, pca_2d.shape[0]):
    if df.classification[i] == 0:
        c1 = pl.scatter(pca_2d[i,0],pca_2d[i,1],c='purple',marker='+')
    elif df.classification[i] == 1:
        c2 = pl.scatter(pca_2d[i,0],pca_2d[i,1],c='cyan',marker='o')
    elif df.classification[i] == 2:
        c3 = pl.scatter(pca_2d[i,0],pca_2d[i,1],c='red',marker='*')
pl.legend([c1, c2, c3], ['C0', 'C1','C2'])
pl.title('Gas consumption dataset with 3 clusters and knownoutcomes')
pl.xlabel('First reduced feature set')
pl.ylabel('Second reduced feature set')
pl.show()
```

3.5 User behavior analysis based on clustering

From the plot above, we able to see that there are three main types of pattern in the dataset: a profile with the typical high morning, noon and night gas consumption (cluster 0), a profile with constant low gas consumption (cluster 1) and finally a profile that has high consumption with no distinct peak hours (cluster 2).

Cluster 0 grouping has 3 distinct peak usage expected at morning, noon and night time. Data has shown that most consumption of gas happen on morning period (06:00 - 09:00) period and second most consumption happen on the night (17:00 - 20:00).

Cluster 1 groupings have a very low utility rate of natural gas. Those households may be the business man or woman who usually do not cook at home. Those households are predicted to be of the working age 35 to 50 and are the main contribution to the country’s economy.

Cluster 2 grouping seems to indicate a relatively high consumption pattern with a householder generally at home throughout the day, whilst not possessing distinct morning and evening demand peaks, and there is not a well delineated minimum in between them. Demographics such as age and employment status in particular regions could contribute to this cluster. For example, most of households may be the elderly or jobless people need to stay at home.

In general, data clustering is used to find similar groups having the same gas consumption patterns. Smart meter data clustering has been widely investigated toward consumers grouping and revealing their energy usage behavior which leads to more efficient tariff policy and tailored energy efficiency programs for specific users.

3.6 Applying price discrimination on different clusters

After computing the clusters of households based on their gas consumption patterns, we can adjust the pricing strategies for different clusters in order to modify their gas consumption behaviours, to limit peak load and to increase natural gas demand during non-peak hours. By doing so, gas companies can reduce the pressure on the natural gas transportation facilities such as the pipelines during peak hours (morning, noon and night as stated in the earlier sections), and the company can possibly reap more benefits by adjusting the pricing strategy.

In [ ]:

```
path = str(Path().resolve().parent)+'\csv'
df = pd.read_csv(path + '/reading_features_classification.csv')
hourly_reading = pd.read_csv(path + '/hourly_readings_with_consumption.csv')

# monthly price of residential gas in Texas from Oct 2015 to Mar 2016
# data retrieved from: https://www.eia.gov/dnav/ng/hist/n3010tx3m.htm
monthly_price = [19.22, 15.52, 9.33, 7.85, 7.87, 10.12]

# compute the monthly consumption for each household and add the columns to store these
 values for future computation
new = hourly_reading['day'].str.split("-", n = 2, expand = True)
hourly_reading.drop(columns = ['day'], inplace = True)
hourly_reading['month'] = new[1]
df = df.assign(oct_consumption = np.random.randn(len(df)), nov_consumption = np.random.
randn(len(df)), dec_consumption = np.random.randn(len(df)), jan_consumption = np.random
.randn(len(df)), feb_consumption = np.random.randn(len(df)), mar_consumption = np.rando
m.randn(len(df)))
for i in range (0,len(df)):
    df.loc[i,'oct_consumption'] = hourly_reading['consumption'][(hourly_reading.dataid
== df.loc[i, 'dataid']) & (hourly_reading.month == '10')].sum()
    df.loc[i,'nov_consumption'] = hourly_reading['consumption'][(hourly_reading.dataid
== df.loc[i, 'dataid']) & (hourly_reading.month == '11')].sum()
    df.loc[i,'dec_consumption'] = hourly_reading['consumption'][(hourly_reading.dataid
== df.loc[i, 'dataid']) & (hourly_reading.month == '12')].sum()
    df.loc[i,'jan_consumption'] = hourly_reading['consumption'][(hourly_reading.dataid
== df.loc[i, 'dataid']) & (hourly_reading.month == '01')].sum()
    df.loc[i,'feb_consumption'] = hourly_reading['consumption'][(hourly_reading.dataid
== df.loc[i, 'dataid']) & (hourly_reading.month == '02')].sum()
    df.loc[i,'mar_consumption'] = hourly_reading['consumption'][(hourly_reading.dataid
== df.loc[i, 'dataid']) & (hourly_reading.month == '03')].sum()
```

The cell below calculates the total cost of natural gas consumed by all the households during the 6 month period from Oct 2015 to Mar 2016 before and after adjusting the pricing strategy.

Instead of changing the price monthly, the average price for the 6 month is calculated as the base amount, and the price for each cluster in different time period will change based on the average price.

Households in cluster 0 (classification == 0) use moderately high amount of gas, and peak of gas consumption happens in the morning (06:00 - 09:00) as well as at night (17:00 - 20:00). The aim will be to reduce their gas consumption in the morning and at night. Threshold values for morning, noon and night consumption are set as the desired gas consumption amount for each household in the given time periods. If the households consume more than the threshold during morning and night, the exceeding amount will be charged more than the average price. Otherwise the price will remain as the average price.

For households in cluster 1 (classification == 1), they generally use very little amount of gas, so the aim is to encourage them to consume more natural gas. Different threshold values, which are lower than the thresholds for cluster 0 and 2, are set, and if the households consume more than the threshold value in certain time periods, the price of natural gas will be lower than the average price.

Households in cluster 2 (classification == 2) consume significantly greater amount of natural gas compared to other households. Therefore the pricing strategy for cluster 2 will be the same as cluster 0, except that the increase in price is much higher, as a small increase in price is unlikely to have much impact on their behaviours. In addition, if they consume less than the threshold values in either of the three time periods, the price will be lower than the average price as an encouragement for the households to consume less.

In [ ]:

```
# calculate and output the original cost of natural gas consumed by all households base
d on the monthly price found in the website
total_cost = 0
for row in df.itertuples():
    total_cost += (row.oct_consumption * monthly_price[0] + row.nov_consumption * month
ly_price[1] + row.dec_consumption * monthly_price[2] + row.jan_consumption * monthly_pr
ice[3] + row.feb_consumption * monthly_price[4] + row.mar_consumption * monthly_price[5
])
print("Original cost of natural gas: %.2f dollars" % (total_cost/1000))

# compute the average price
total_consumption = 0
for row in df.itertuples():
    total_consumption += row.totalConsumption
avg_price = total_cost / total_consumption

# set threshold values
morningThreshold = 3000
noonThreshold = 2400
nightThreshold = 4200

morningThreshold_2 = 2400
noonThreshold_2 = 1800
nightThreshold_2 = 3000

# compute the new estimated cost after changing the pricing strategy
total_adjusted_cost = 0
for row in df.itertuples():
    if (row.classification == 0):
        if (row.morningTotal > morningThreshold):
            total_adjusted_cost += (row.morningTotal - morningThreshold) * (avg_price +
 2) + morningThreshold * avg_price
        else:
            total_adjusted_cost += row.morningTotal * avg_price
        total_adjusted_cost += row.noonTotal * avg_price
        if (row.nightTotal > nightThreshold):
            total_adjusted_cost += (row.nightTotal - nightThreshold) * (avg_price + 2)
+ nightThreshold * avg_price
        else:
            total_adjusted_cost += row.nightTotal * avg_price

    if (row.classification == 1):
        if (row.morningTotal > morningThreshold_2):
            total_adjusted_cost += row.morningTotal * (avg_price - 1)
        else:
            total_adjusted_cost += row.morningTotal * avg_price
        if (row.noonTotal > noonThreshold_2):
            total_adjusted_cost += row.noonTotal * (avg_price - 1)
        else:
            total_adjusted_cost += row.noonTotal * avg_price
        if (row.nightTotal > nightThreshold_2):
            total_adjusted_cost += row.nightTotal * (avg_price - 1)
        else:
            total_adjusted_cost += row.nightTotal * avg_price

    if (row.classification == 2):
        if (row.morningTotal > morningThreshold):
            total_adjusted_cost += (row.morningTotal - morningThreshold) * (avg_price +
 4) + morningThreshold * avg_price
        else:
            total_adjusted_cost += row.morningTotal * (avg_price - 1)
        if (row.noonTotal > noonThreshold):
            total_adjusted_cost += (row.noonTotal - noonThreshold) * (avg_price + 4) +
noonThreshold * avg_price
        else:
            total_adjusted_cost += row.noonTotal * (avg_price - 1)
        if (row.nightTotal > nightThreshold):
            total_adjusted_cost += (row.nightTotal - nightThreshold) * (avg_price + 4)
+ nightThreshold * avg_price
        else:
            total_adjusted_cost += row.nightTotal * (avg_price - 1)

    total_adjusted_cost += (row.totalConsumption - row.morningTotal - row.noonTotal - r
ow.nightTotal) * avg_price
print("Estimated cost after changing the price: %.2f dollars" % (total_adjusted_cost/10
00))
```

3.7 Conclusion

In our project, investigations have been performed into using clustering methodsin data mining time-series data from smart meters. The problem is to identify patterns and trends in gas consumption profiles of residential customers over a large period of time, and group similar profiles into 3 clusters. The results has shown accurate grouping of accounts similar in their energy usage patterns, and potential for adjusting the pricing strategies for different clusters in order to modify their gas consumption behaviours for higher profits and better energy efficiency.