MEASURE FOR ENERGY CONSUMPTION

PHASE 4:DEVELOPMENT (PART 2)

INTRODUCTION:

Energy consumption is a critical aspect in many fields, from household appliances to industrial processes, and even in the realm of artificial intelligence. It refers to the amount of energy or power that is consumed by a device, system, process, or facility over a certain period of time.

Understanding and measuring energy consumption can lead to more efficient use of resources, cost savings, and reduced environmental impact. It's a complex field with many variables, but with careful measurement and management, significant improvements can be made.

MODEL TRAINING:

Understanding the Problem: The first step is to understand the problem and the data you have. This includes understanding what factors might affect energy consumption and how they can be quantified and measured.

Data Collection: Collect data related to energy consumption and the factors that affect it. This could include data on usage patterns, environmental conditions, device specifications, etc.

Preprocessing: Clean the data and handle missing values or outliers. This step also involves feature engineering, where you create new features from the existing data to help improve the model's performance.

Model Selection: Choose a machine learning model that's suitable for your problem. Linear regression models, decision trees, random forests, and gradient boosting models are commonly used for prediction tasks[1].

Training: Train your model on your preprocessed dataset.

Evaluation: Evaluate your model's performance on a separate test set.

Optimization: Based on your evaluation, optimize your model by tuning its parameters or by choosing a different model.

Deployment: Once you're satisfied with your model's performance, deploy it to start making predictions on real-world data.

It's important to note that predicting energy consumption is not straightforward due to the many factors that can influence it. For example, the number of CPU operations (FLOPs) required by machine learning models can be used to predict their energy consumption[2]. However, other factors such as the type of machine used for inference also need to be considered[2].

There are also efforts in the AI community towards "Green AI", which aims to achieve state-of-the-art results while consuming less energy. This involves studying both the training and inference phases of AI systems' life cycles.

For more detailed information, you might find these resources helpful: "Predicting and Reducing Energy Consumption of Machine Learning Models" and "How to Measure Energy Consumption in Machine Learning Algorithms". There are also online courses available on this topic.

Code:

```
#Library importingimport pandas as pdimport numpy as npimport matplotlib.
pyplot as plt

import statsmodels.api as smfrom scipy import statsimport itertools

from sklearn.cluster import KMeansfrom sklearn.preprocessing import MinM
axScalerfrom sklearn.metrics import silhouette_samples, silhouette_scorefro
m sklearn.metrics import mean_squared_error

import datetimeimport osimport mathimport gc
```

Data extraction

The data used for this notebook is separated into different files/blocks and needs to be assembled into a data frame first.

In [2]:

```
def data_extraction(path):

    dataframe = pd.DataFrame()

    for i in os.listdir(path):

        df_temp = pd.read_csv(str(path) + "/" + str(i))
        df_temp = df_temp[["LCLid","day","energy_sum"]]
        df_temp.reset_index()
        dataframe = dataframe.append(df_temp)

    return dataframe
```

In [3]:

```
path = "../input/smart-meters-in-london/daily_dataset/daily_dataset"df = data
_extraction(path)

del path
```

The core metric we will try to predict is the mean energy per unique household. We need to calculate it as it isn't given initially in the data. To this end, we will first take the daily sum of energy consumption and divide it by the number of unique households on the same day. Its important to take unique households as the number varies per day.

In [4]:

```
### Energy per Household###

energy = df.groupby("day")[["energy_sum"]].sum()count_of_house = df.grou
pby("day")[["LCLid"]].nunique()

df_energy = energy.merge(count_of_house, on="day").reset_index()
```

```
df_energy["energy_per_household"] = df_energy["energy_sum"] / df_energy
["LCLid"]df_energy["day"] = pd.to_datetime(df_energy["day"])

del energy, count_of_house

gc.collect()
```

Out[4]:

0

Importing the weather and holiday datasets. The holidays might be an interesting metric to look into further. They might have a different impact depending if the meter is placed on a household or business building. The data we are using originates from households so we might see an increase depending on the holiday.

In [5]:

```
#Weather and holiday dataweather_df = pd.read_csv("../input/smart-meters-i
n-london/weather_daily_darksky.csv")holiday_df = pd.read_csv("../input/sma
rt-meters-in-london/uk_bank_holidays.csv")
```

Clustering

The first step is to prepare the weather dataset for clustering. We will do this by filtering out some features and creating a new data frame. It is also important to convert the datatype of the "time" column into datetime.

In [6]:

```
weather_df = weather_df[["temperatureMax",

                        "windBearing",

                        "dewPoint",

                        "cloudCover",

                        "windSpeed",

                        "pressure",

                        "time",

                        "humidity"]]

weather_df["time"] = pd.to_datetime(weather_df["time"])

weather_df.dropna(inplace=True)
```

Looking at the correlations we get a general idea of what weather data can be clustered. This was a bit of an experimental step and this is the best version I got. I also decided not to include temperature into the clustering portion as the column was too important and I would rather not lose information on it.

In [7]:

```
weather_df.corr()
```

Out[7]:

	temperatureMax	windBearing	dewPoint	cloudCover	windSpeed	pressure	humidity
temperatureMax	1.000000	0.066226	0.854893	-0.332584	-0.147009	0.122966	-0.399969
windBearing	0.066226	1.000000	0.087704	-0.083740	0.078558	-0.030625	0.001558
dewPoint	0.854893	0.087704	1.000000	-0.003382	-0.090370	-0.026797	0.079938
cloudCover	-0.332584	-0.083740	-0.003382	1.000000	0.165238	-0.101524	0.492810
windSpeed	-0.147009	0.078558	-0.090370	0.165238	1.000000	-0.333642	-0.056839
pressure	0.122966	-0.030625	-0.026797	-0.101524	-0.333642	1.000000	-0.240828
humidity	-0.399969	0.001558	0.079938	0.492810	-0.056839	-0.240828	1.000000

An important step at this point is to scale the data. This is to ensure that all the columns are valued equally in the K-mean clustering step.

In [8]:

```
scaler = MinMaxScaler()weather_scaled = scaler.fit_transform(weather_df[["cloudCover","humidity","windSpeed"]]).astype("float64")
```

In [9]:

```
kmeans_kwargs = {"init": "k-means++",
                 "n_init": 10,
                 "max_iter": 450,
                 "random_state": 42}
def clustering (df):
```

```
    """    Tests posible k_mean cluster instances and scores them based on t
he silhouette score    """

    sc = []



    for k in range(2,15):



        kmeans = KMeans(n_clusters=k, **kmeans_kwargs)

        kmeans.fit_transform(df)

        score = silhouette_score(df,kmeans.labels_)

        sc.append(score)



    return sc
```

The optimal number of clusters can be determined by graphing the silhouette score. **Silhouette Coefficiency** measures how similar an object is to its own cluster. It's a great tool to determine the optimal amount of clusters.

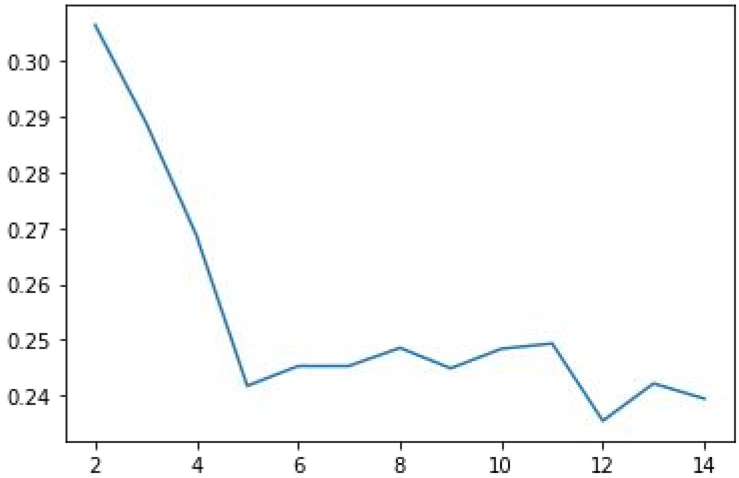
The best practice is to take the "L" part of the curvature as the optimal number of clusters.

In [10]:

```
sc = clustering(weather_scaled)plt.plot(range(2, 15), sc)
```

Out[10]:

```
[<matplotlib.lines.Line2D at 0x7fac5c84c350>]
```



In [11]:

```
#Creating the KMean that will be used and droping the unused weather data
kmeans = KMeans(n_clusters=5, **kmeans_kwargs)weather_df["Clusters"]=
 kmeans.fit(weather_scaled).labels_

to_drop = ["windBearing","dewPoint","cloudCover","windSpeed","pressure","
humidity"]

weather_df.drop(to_drop,axis=1,inplace=True)
```

Additional data preparation

This portion contains some additional data preparation before we can go on to SARIMAX predictions.

The holiday column will be coded based on a binary system, with the 1 representing that the date has a holiday and that the date is without a holiday.

In [12]:

```
holiday_df["Bank holidays"] = pd.to_datetime(holiday_df["Bank holidays"])
```

In [13]:

```
df_energy = df_energy.merge(weather_df, left_on="day",right_on="time")fin
al_df = df_energy.merge(holiday_df, left_on = "day",right_on = "Bank holiday
s",how = 'left')final_df["holiday_id"] = np.where(final_df['Bank holidays'].isna(),
0,1)
```

In [14]:

```
final_df.head()
```

Out[14]:

	day	energy_sum	LCLid	energy_per_household	temperatureMax	time	Clusters	Bank holidays	Type	holiday_id

	day	energy_sum	LCLid	energy_per_household	temperatureMax	time	Clusters	Bank holidays	Type	holiday_id
0	2011-11-23	90.385000	13	6.952692	10.36	2011-11-23	0	NaT	NaN	0
1	2011-11-24	213.412000	25	8.536480	12.93	2011-11-24	0	NaT	NaN	0
2	2011-11-25	303.993000	32	9.499781	13.03	2011-11-25	1	NaT	NaN	0
3	2011-11-26	420.976000	41	10.267707	12.96	2011-11-26	4	NaT	NaN	0
4	2011-11-27	444.883001	41	10.850805	13.54	2011-11-27	1	NaT	NaN	0

In [15]:

```
to_drop = ["energy_sum","LCLid","time","Bank holidays","Type"]

final_df.drop(to_drop, axis=1, inplace=True)
```

In [16]:

```
final_df.head()
```

Out[16]:

	day	energy_per_household	temperatureMax	Clusters	holiday_id
0	2011-11-23	6.952692	10.36	0	0
1	2011-11-24	8.536480	12.93	0	0
2	2011-11-25	9.499781	13.03	1	0
3	2011-11-26	10.267707	12.96	4	0
4	2011-11-27	10.850805	13.54	1	0

Finalizing the data to be used and splitting it into train/test portions.

In [17]:

```
final_df.index = pd.DatetimeIndex(final_df["day"]).to_period("D")

model_data = final_df[["energy_per_household","temperatureMax","Clusters","holiday_id"]]
```

```
train = model_data.iloc[0:len(model_data)-30] test = model_data.iloc[len(trai
n):len(model_data)]

del model_data
```

In [18]:

```
train.head()
```

Out[18]:

	energy_per_household	temperatureMax	Clusters	holiday_id
day				
2011-11-23	6.952692	10.36	0	0
2011-11-24	8.536480	12.93	0	0
2011-11-25	9.499781	13.03	1	0
2011-11-26	10.267707	12.96	4	0
2011-11-27	10.850805	13.54	1	0

In [19]:

```
###SARIMAX###

#Constructs all possible parameter combinations.p = d = q = range(0,2)pdq
= list(itertools.product(p,d,q))

seasonal_pdq = [(x[0],x[1],x[2],12) for x in list(itertools.product(p,d,q))]
```

SARIMAX

We will use SARIMAX to predict the mean consumption of the dataset. However, before doing that we need to test out the optimal pqd combination of the model. I will use a very brute force method for his as the dataset isn't that large.

In [20]:

```
def sarimax_function(endog,exog,pdq,s_pdq):



    """    The function uses a brute force approach to apply all possible pdq co
mbinations and evaluate the model    """



    result_list = []

    for param in pdq:
```

```
        for s_param in s_pdq:



            model = sm.tsa.statespace.SARIMAX(endog=endog,exog=exog, ord
er=param, seasonal_order=s_param,

            enforce_invertibility=False,enforce_stationarity=True)



            results = model.fit()

            result_list.append([param,s_param,results.aic])

            #print("ARIMA Parameters: {} x: {}. AIC: {}".format(param,s_param,re
sults.aic))



    return result_list,results
```

When using a SARIMAX predictore we need to define the endog and exog variables to successfully run the model. To explain the two in simple terms:

- The endog variable is the target variable or the response variable or the model.
- The exog variable is the independent variable designed to explain the endog variable.

In [21]:

```
endog = train["energy_per_household"]exog = train[["Clusters","holiday_id","
temperatureMax"]]
```

In [22]:

```
result_list,results = sarimax_function(endog,exog,pdq,seasonal_pdq)
```

```
/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568: ConvergenceWa
rning: Maximum Likelihood optimization failed to converge. Check mle_retvals

  "Check mle_retvals", ConvergenceWarning)

/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568: ConvergenceWa
rning: Maximum Likelihood optimization failed to converge. Check mle_retvals

  "Check mle_retvals", ConvergenceWarning)

/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568: ConvergenceWa
rning: Maximum Likelihood optimization failed to converge. Check mle_retvals
```

"Check mle_retvals", ConvergenceWarning)

/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568: ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals

"Check mle_retvals", ConvergenceWarning)

The results of the test indicate the optimal pdq combination based on AIC. AIC (Akaike Information Criterion -> AIC=ln (sm2) + 2m/T). As a model selection tool, AIC has some limitations as it only provides a relative evaluation of the model. However, it is an excellent metric for checking the general quality of a model.

In [23]:

```
results_dataframe = pd.DataFrame(result_list, columns=["dpq","s_dpq","aic
"]).sort_values(by="aic")results_dataframe.head()
```

Out[23]:

	dpq	s_dpq	aic
61	(1, 1, 1)	(1, 0, 1, 12)	606.477509
56	(1, 1, 1)	(0, 0, 0, 12)	610.586427
57	(1, 1, 1)	(0, 0, 1, 12)	610.695748
60	(1, 1, 1)	(1, 0, 0, 12)	611.089445
29	(0, 1, 1)	(1, 0, 1, 12)	633.113746

Prediction

We first need to generate a model based on the information we gathered in this notebook and "train" it on the training portion of the data.

In [24]:

```
model = sm.tsa.statespace.SARIMAX(endog=endog,exog=exog, order=(1, 1,
 1), seasonal_order=(1, 0, 1, 12),

        enforce_invertibility=False,enforce_stationarity=True).fit()

print(model.summary().tables[1])
```

```
==============================================================================
               coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Clusters       0.0335      0.018      1.867      0.062      -0.002       0.069
holiday_id    -0.0661      0.292     -0.226      0.821      -0.639       0.507
```

temperatureMax	-0.1225	0.013	-9.643	0.000	-0.147	-0.098
ar.L1	0.4408	0.077	5.703	0.000	0.289	0.592
ma.L1	-1.2447	0.082	-15.088	0.000	-1.406	-1.083
ar.S.L12	0.7533	0.132	5.694	0.000	0.494	1.013
ma.S.L12	-0.8679	0.109	-7.947	0.000	-1.082	-0.654
sigma2	0.1856	0.029	6.472	0.000	0.129	0.242

```
==============================================================================
/opt/conda/lib/python3.7/site-packages/statsmodels/base/model.py:568: ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals

  "Check mle_retvals", ConvergenceWarning)
```

Defining the test exog variables

In [25]:

```
exog = test[["Clusters","holiday_id","temperatureMax"]]
```

Predicting and storing the data in a data frame for comparison.

In [26]:

```
predict = model.predict(start = len(train),end = len(train)+len(test)-1,
                        exog = test[["Clusters","holiday_id","temperatureMax"]])
test["prediction"] = predict.values
```

We will test the prediction using MAE (Mean absolute error) and Mean squared error to get a general idea of how good the model is.

In [27]:

```
test["diff"] = test["energy_per_household"] - test["prediction"]results = mean_squared_error(test["energy_per_household"],test["prediction"])print(results)
```

```
4.802925839555962
```

In [28]:

```
MAE = test['diff'].sum()/len(test)print(MAE)
```

```
-0.672271822654676
```

The results are generally pretty ok. However, I noticed that there is an outlier in one day so we will also take a look at it.

In [29]:

```
copy_test = test.copy()
```

In [30]:

```
copy_test.sort_values(by=["diff"])
```

Out[30]:

	energy_per_household	temperatureMax	Clusters	holiday_id	prediction	diff
day						
2014-02-28	0.208997	7.35	2	0	11.844842	-11.635845
2014-02-27	10.356350	10.31	1	0	11.512883	-1.156533
2014-02-25	10.294997	11.43	4	0	11.414887	-1.119890
2014-02-26	10.202945	11.29	1	0	11.302566	-1.099620
2014-02-21	10.518126	10.15	1	0	11.472630	-0.954503
2014-02-19	10.674624	10.13	2	0	11.402173	-0.727550
2014-02-20	10.573835	12.50	4	0	11.238258	-0.664422
2014-02-03	11.280011	7.99	1	0	11.846590	-0.566578
2014-02-18	10.781898	10.13	0	0	11.344515	-0.562617
2014-02-24	10.411403	14.23	1	0	10.967259	-0.555856
2014-02-04	11.095584	8.88	1	0	11.625591	-0.530007
2014-02-13	11.285737	7.37	1	0	11.813931	-0.528195
2014-02-07	10.972318	9.81	2	0	11.409018	-0.436700
2014-02-22	10.776242	11.63	1	0	11.202838	-0.426596
2014-02-17	10.979566	10.67	2	0	11.300639	-0.321073
2014-02-11	11.452649	7.51	1	0	11.704821	-0.252172
2014-02-10	11.264175	8.78	0	0	11.492119	-0.227944
2014-02-15	11.490470	9.90	4	0	11.685800	-0.195331
2014-	11.685169	5.94	2	0	11.846024	-0.160855

	energy_per_household	temperatureMax	Clusters	holiday_id	prediction	diff
day						
01-30						
2014-02-08	11.569300	9.13	4	0	11.638319	-0.069019
2014-02-12	11.679099	8.83	4	0	11.730167	-0.051068
2014-02-05	11.415105	9.64	4	0	11.449554	-0.034450
2014-02-06	11.445403	9.81	2	0	11.421321	0.024082
2014-02-16	11.582159	9.98	3	0	11.557253	0.024906
2014-02-01	11.710582	9.72	1	0	11.517657	0.192925
2014-02-23	11.480411	11.94	4	0	11.283133	0.197278
2014-01-31	11.857957	8.83	2	0	11.638921	0.219036
2014-02-09	12.202967	8.18	4	0	11.817062	0.385905
2014-02-14	11.816914	12.02	4	0	11.306248	0.510666
2014-02-02	12.078164	9.30	1	0	11.524295	0.553870

In [31]:

```
### Results without the outlier ###

results = mean_squared_error(copy_test.iloc[:-1,:]["energy_per_household"],
copy_test.iloc[:-1,:]["prediction"])print(results)
```

0.29982332169098413

In [32]:

```
MAE =copy_test.iloc[:-1,:]["diff"].sum()/len(test)print(MAE)
```

-0.2844103083402664

The same metrics without the outlier look a lot better!

Conclusion

This was a small project on clustering and I see myself using this in specific situations. The speed we gain when during this might not outweigh the small decrease in accuracy when predicting consumption, but its an interesting alternative for budgeting larger scale portofolios.