Introduction

This is the ***third*** of a series where I look at big datasets, and in each case I'm using a different tool to carry out the same analysis on the same dataset.

This time I'm using **PySpark**, the Python API for **Apache Spark**, an open source, distributed computing framework and set of libraries for real-time, large-scale data processing. You can find each notebook in the series in my Github repo, including:

1. Pandas chunksize
2. Dask library
3. PySpark

There is a little more explanation in the first notebook (Pandas chunksize) on the overall approach to the analysis. In the other notebooks I focus more on the elements specific to the tool being used.

Dataset description

Throughout the series we'll use the SmartMeter Energy Consumption Data in London Households dataset, which according to the website contains:

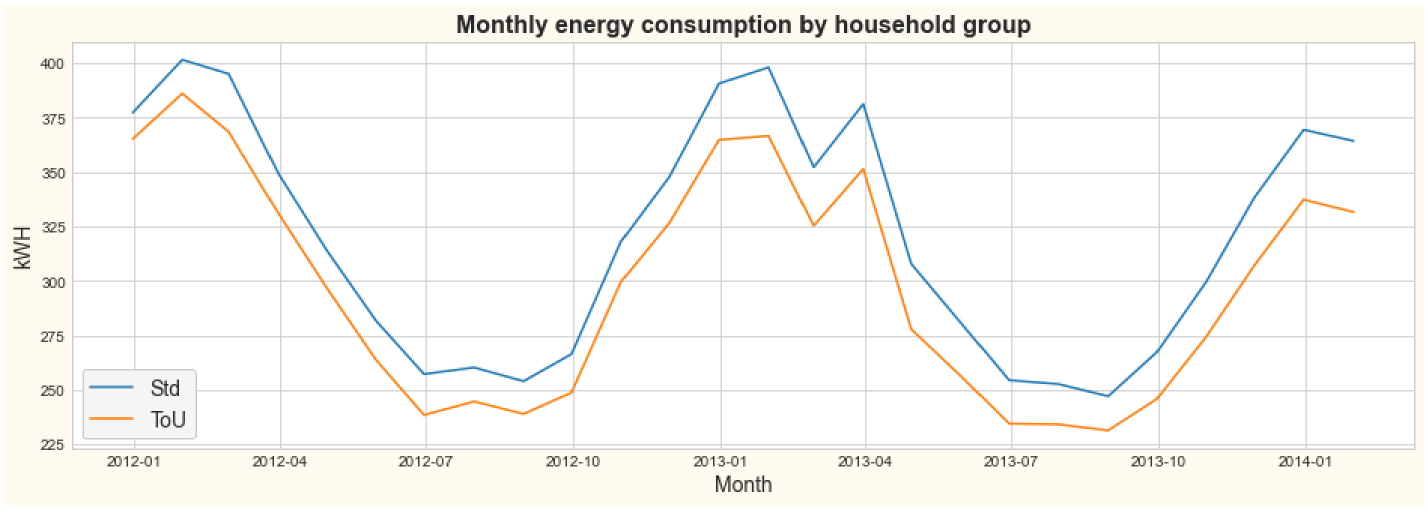
> Energy consumption readings for a sample of 5,567 London Households that took part in the UK Power Networks led Low Carbon London project between November 2011 and February 2014.

The households were divided into two groups:

- Those who were sent Dynamic Time of Use (dToU) energy prices (labelled "High", "Medium", or "Low") a day in advance of the price being applied.
- Those who were subject to the Standard tariff.

One aim of the study was to see if pricing knowledge would affect energy consumption behaviour.

Results



The results show the expected seasonal variation with a clear difference between the two groups, suggesting that energy price knowledge does indeed help reduce energy consumption.

The rest of the notebook shows how this chart was produced from the raw data.

Introduction to PySpark and Apache Spark

A few phrases to explain the basics (taken from here).

> Apache Spark is basically a computational engine that works with huge sets of data by processing them in parallel and batch systems. Spark is written in Scala, and PySpark was released to support the collaboration of Spark and Python. In addition to providing an API for Spark, PySpark helps you interface with Resilient Distributed Datasets (RDDs) by leveraging the Py4j library.
>
> The key data type used in PySpark is the Spark dataframe. This object can be thought of as a table distributed across a cluster, and has functionality that is similar to dataframes in R and Pandas. If you want to do distributed computation using PySpark, then you'll need to perform operations on Spark dataframes and not other Python data types.
>
> One of the key differences between Pandas and Spark dataframes is eager versus lazy execution. In PySpark, operations are delayed until a result is actually requested in the pipeline. For example, you can specify operations for loading a data set from Amazon S3 and applying a number of transformations to the dataframe, but these operations won't be applied immediately. Instead, a graph of transformations is recorded, and once the data are actually needed, for example when writing the results back to S3, then the transformations are applied as a single pipeline operation. This approach is used to avoid pulling the full dataframe into memory, and enables more effective processing across a cluster of machines. With Pandas dataframes, everything is pulled into memory, and every Pandas operation is applied immediately.

In summary then, we'll be using PySpark in much the same way we used the Dask library in that:

- We can use multiple CPUs on a single machine instead of just one.
- Operations are "lazy", running only when necessary.
- We use RDDs (Resilient Distributed Datasets) instead of a Pandas dataframe to manipulate the data.

Installation

To install and set up PySpark I followed the advice here, using the second method (FindSpark).

Accessing the data

The data is downloadable as a single zip file which contains a csv file of 167 million rows. If the `curl` command doesn't work (and it will take a while as it's a file of 800MB), you can download the file here and put it in the folder `data` which is in the folder where this notebook is saved.

In [ ]:
```
!curl "https://data.london.gov.uk/download/smartmeter-energy-use-data-in-london-households/3527bf39-d93e-4071-8451-df2ade1ea4f2/LCL-FullData.zip" --location --create-dirs -o "data/LCL-FullData.zip"
```

First we unzip the data. This may take a while! Alternatively you can unzip it manually using whatever unzip utility you have. Just make sure the extracted file is in a folder called `data` within the folder where your notebook is saved.

In [ ]:
```
!unzip "data/LCL-FullData.zip" -d "data"
```

Examining the data

In [1]:
```
import findspark
findspark.init()
import pyspark
```

In [2]:
```
import pandas as pd
```

First we must create a `SparkContext`.

In [3]:
```
sc = pyspark.SparkContext(appName="LSE")
```

```
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
22/06/27 09:24:48 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
```

Now let's load the data into an RDD that we'll call `raw_data`.

In [4]:
```
raw_data = sc.textFile("data/CC_LCL-FullData.csv")
```

In [5]:
```
raw_data.take(15)
```

Out[5]:
```
['LCLid,stdorToU,DateTime,KWH/hh (per half hour) ',
 'MAC000002,Std,2012-10-12 00:30:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 01:00:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 01:30:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 02:00:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 02:30:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 03:00:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 03:30:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 04:00:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 04:30:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 05:00:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 05:30:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 06:00:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 06:30:00.0000000, 0 ',
 'MAC000002,Std,2012-10-12 07:00:00.0000000, 0 ']
```

Note that our data is a list, each element being a single line of data in a single string.

We can also see that our data is divided into 255 partitions, by using the method `getNumPartitions`.

In [6]:
```
raw_data.getNumPartitions()
```

Out[6]:
```
255
```

Let's remove the column headers from the data as we will be applying functions to our RDD and we want to apply them to the data, not the headers.

We use the `filter` method to remove the row.

In [7]:
```
header = raw_data.first()
full_data = raw_data.filter(lambda x : x != header)
full_data.take(15)
```

```
Out[7]: ['MAC000002,Std,2012-10-12 00:30:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 01:00:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 01:30:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 02:00:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 02:30:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 03:00:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 03:30:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 04:00:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 04:30:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 05:00:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 05:30:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 06:00:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 06:30:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 07:00:00.0000000, 0 ',
         'MAC000002,Std,2012-10-12 07:30:00.0000000, 0 ']
```

Cleaning the data

Let's work on a small subset of the data (10,000 rows) to develop each processing step.

```
In [8]: test_data = sc.parallelize(full_data.take(10000))
        test_data.getNumPartitions()
```

```
Out[8]: 8
```

Before we do anything else let's remove any duplicates.

```
In [9]: test_data_no_dupes = test_data.distinct()
```

As expected, the operation has not been executed yet because we are in "lazy" execution mode. To see the results of our operation we use the `collect` method.

```
In [10]: len(test_data_no_dupes.collect())
```

```
Out[10]: 9993
```

That's worked - 7 duplicates removed. Now let's move onto processing.

We'll be using the `map` method to process the data. The first step is to split each string (representing a line) into a list.

```
In [11]: def split_line_into_list(x):
             '''
             Split single string of line in csv format into values corresponding to columns
             '''
             return x.split(",")
```

```
In [12]: split_test_data = test_data_no_dupes.map(split_line_into_list)
```

```
In [13]: split_test_data.collect()[:15]
```

```
Out[13]: [['MAC000002', 'Std', '2012-10-12 17:00:00.0000000', ' 0.493 '],
          ['MAC000002', 'Std', '2012-10-12 20:00:00.0000000', ' 0.198 '],
          ['MAC000002', 'Std', '2012-10-13 01:00:00.0000000', ' 0.275 '],
          ['MAC000002', 'Std', '2012-10-13 02:00:00.0000000', ' 0.211 '],
          ['MAC000002', 'Std', '2012-10-13 03:30:00.0000000', ' 0.119 '],
          ['MAC000002', 'Std', '2012-10-13 09:30:00.0000000', ' 0.191 '],
          ['MAC000002', 'Std', '2012-10-13 12:00:00.0000000', ' 0.076 '],
          ['MAC000002', 'Std', '2012-10-13 12:30:00.0000000', ' 0.133 '],
          ['MAC000002', 'Std', '2012-10-13 13:30:00.0000000', ' 0.133 '],
          ['MAC000002', 'Std', '2012-10-13 16:30:00.0000000', ' 0.184 '],
          ['MAC000002', 'Std', '2012-10-13 19:30:00.0000000', ' 0.278 '],
          ['MAC000002', 'Std', '2012-10-13 22:30:00.0000000', ' 0.188 '],
          ['MAC000002', 'Std', '2012-10-14 00:00:00.0000000', ' 0.262 '],
          ['MAC000002', 'Std', '2012-10-14 10:00:00.0000000', ' 0.524 '],
          ['MAC000002', 'Std', '2012-10-14 12:30:00.0000000', ' 0.22 ']]
```

Now we can carry out the cleaning operations (the same as in the other notebooks in the series):

- Convert the kWh data to numeric
- Convert the timestamp data to date format, ready for grouping

```
In [14]: def convert_kwh_data_to_numeric(x):
             '''
             Convert the kWh value from string to float
             '''
             x[3] = float(x[3])
             return x
```

```
In [15]: numeric_kwh_test_data = split_test_data.map(convert_kwh_data_to_numeric)
         numeric_kwh_test_data.collect()
```

```
22/06/27 09:26:51 ERROR Executor: Exception in task 2.0 in stage 9.0 (TID 30)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 619, in main
    process()
  File "/opt/spark/python/lib/pyspark.zip/pyspark/worker.py", line 611, in process
    serializer.dump_stream(out_iter, outfile)
  File "/opt/spark/python/lib/pyspark.zip/pyspark/serializers.py", line 259, in dump_stream
    vs = list(itertools.islice(iterator, batch))
  File "/opt/spark/python/lib/pyspark.zip/pyspark/util.py", line 74, in wrapper
    return f(*args, **kwargs)
  File "/tmp/ipykernel_85793/95641023.py", line 5, in convert_kwh_data_to_numeric
ValueError: could not convert string to float: 'Null'
```

The error message tells us that a string value of `"Null"` is preventing the conversion. So let's first remove those rows with the `filter` method.

In [16]:
```
def remove_nulls(x):
    return x[3] != 'Null'
```

In [17]:
```
filtered_test_data = split_test_data.filter(remove_nulls)
```

In [18]:
```
numeric_kwh_test_data = filtered_test_data.map(convert_kwh_data_to_numeric)
numeric_kwh_test_data.collect()[:15]
```

Out[18]:
```
[['MAC000002', 'Std', '2012-10-12 17:00:00.0000000', 0.493],
 ['MAC000002', 'Std', '2012-10-12 20:00:00.0000000', 0.198],
 ['MAC000002', 'Std', '2012-10-13 01:00:00.0000000', 0.275],
 ['MAC000002', 'Std', '2012-10-13 02:00:00.0000000', 0.211],
 ['MAC000002', 'Std', '2012-10-13 03:30:00.0000000', 0.119],
 ['MAC000002', 'Std', '2012-10-13 09:30:00.0000000', 0.191],
 ['MAC000002', 'Std', '2012-10-13 12:00:00.0000000', 0.076],
 ['MAC000002', 'Std', '2012-10-13 12:30:00.0000000', 0.133],
 ['MAC000002', 'Std', '2012-10-13 13:30:00.0000000', 0.133],
 ['MAC000002', 'Std', '2012-10-13 16:30:00.0000000', 0.184],
 ['MAC000002', 'Std', '2012-10-13 19:30:00.0000000', 0.278],
 ['MAC000002', 'Std', '2012-10-13 22:30:00.0000000', 0.188],
 ['MAC000002', 'Std', '2012-10-14 00:00:00.0000000', 0.262],
 ['MAC000002', 'Std', '2012-10-14 10:00:00.0000000', 0.524],
 ['MAC000002', 'Std', '2012-10-14 12:30:00.0000000', 0.22]]
```

That's working fine now. Now for the date conversion.

In [19]:
```
def convert_timestamp_to_date_string(x):
    '''
    Convert the timestamp into a date string
    '''
    x[2] = x[2].split(" ")[0]
    return x
```

In [20]:
```
cleaned_test_data = numeric_kwh_test_data.map(convert_timestamp_to_date_string)
cleaned_test_data.collect()[:15]
```

```
Out[20]: [['MAC000002', 'Std', '2012-10-12', 0.493],
          ['MAC000002', 'Std', '2012-10-12', 0.198],
          ['MAC000002', 'Std', '2012-10-13', 0.275],
          ['MAC000002', 'Std', '2012-10-13', 0.211],
          ['MAC000002', 'Std', '2012-10-13', 0.119],
          ['MAC000002', 'Std', '2012-10-13', 0.191],
          ['MAC000002', 'Std', '2012-10-13', 0.076],
          ['MAC000002', 'Std', '2012-10-13', 0.133],
          ['MAC000002', 'Std', '2012-10-13', 0.133],
          ['MAC000002', 'Std', '2012-10-13', 0.184],
          ['MAC000002', 'Std', '2012-10-13', 0.278],
          ['MAC000002', 'Std', '2012-10-13', 0.188],
          ['MAC000002', 'Std', '2012-10-14', 0.262],
          ['MAC000002', 'Std', '2012-10-14', 0.524],
          ['MAC000002', 'Std', '2012-10-14', 0.22]]
```

Aggregating the data

To aggregate we're going to use the `reduceByKey` method. For that we need to restructure our data into tuples in the form `(key, data_to_reduce)`. In our case we want the key to be a combined key of date, household id and tariff type, and the data to be the kWh value. So our key will also be a tuple.

```
In [21]: def create_keyed_kwh_data(x):
             return ( (x[0], x[1], x[2]), x[3] )
```

```
In [22]: keyed_kwh_data = cleaned_test_data.map(create_keyed_kwh_data)
         keyed_kwh_data.collect()[:15]
```

```
Out[22]: [(('MAC000002', 'Std', '2012-10-12'), 0.493),
          (('MAC000002', 'Std', '2012-10-12'), 0.198),
          (('MAC000002', 'Std', '2012-10-13'), 0.275),
          (('MAC000002', 'Std', '2012-10-13'), 0.211),
          (('MAC000002', 'Std', '2012-10-13'), 0.119),
          (('MAC000002', 'Std', '2012-10-13'), 0.191),
          (('MAC000002', 'Std', '2012-10-13'), 0.076),
          (('MAC000002', 'Std', '2012-10-13'), 0.133),
          (('MAC000002', 'Std', '2012-10-13'), 0.133),
          (('MAC000002', 'Std', '2012-10-13'), 0.184),
          (('MAC000002', 'Std', '2012-10-13'), 0.278),
          (('MAC000002', 'Std', '2012-10-13'), 0.188),
          (('MAC000002', 'Std', '2012-10-14'), 0.262),
          (('MAC000002', 'Std', '2012-10-14'), 0.524),
          (('MAC000002', 'Std', '2012-10-14'), 0.22)]
```

Now we use `reduceByKey` to sum the kWh values.

```
In [23]: aggregated_test_data = keyed_kwh_data.reduceByKey(lambda x, y : x + y)
         aggregated_test_data.collect()[:15]
```

```
Out[23]: [(('MAC000002', 'Std', '2012-10-12'), 7.098000000000001),
          (('MAC000002', 'Std', '2012-10-18'), 10.750999999999998),
          (('MAC000002', 'Std', '2012-10-24'), 15.5370001),
          (('MAC000002', 'Std', '2012-10-25'), 13.128),
          (('MAC000002', 'Std', '2012-11-01'), 12.209),
          (('MAC000002', 'Std', '2012-11-03'), 14.347),
          (('MAC000002', 'Std', '2012-11-10'), 13.245000000000001),
          (('MAC000002', 'Std', '2012-11-12'), 12.321000000000002),
          (('MAC000002', 'Std', '2012-11-13'), 10.264000000000001),
          (('MAC000002', 'Std', '2012-11-22'), 9.44),
          (('MAC000002', 'Std', '2012-12-08'), 13.441999999999998),
          (('MAC000002', 'Std', '2012-12-28'), 11.157),
          (('MAC000002', 'Std', '2013-01-01'), 10.8),
          (('MAC000002', 'Std', '2013-01-06'), 10.293000000000001),
          (('MAC000002', 'Std', '2013-01-11'), 10.978999899999996)]
```

```
In [24]: aggregated_test_data_table = aggregated_test_data.map(lambda x : (x[0][0], x[0][1], x[0][2],
         x[1]))
         aggregated_test_data_table.collect()[:15]
```

```
Out[24]: [('MAC000002', 'Std', '2012-10-12', 7.098000000000001),
          ('MAC000002', 'Std', '2012-10-18', 10.750999999999998),
          ('MAC000002', 'Std', '2012-10-24', 15.5370001),
          ('MAC000002', 'Std', '2012-10-25', 13.128),
          ('MAC000002', 'Std', '2012-11-01', 12.209),
          ('MAC000002', 'Std', '2012-11-03', 14.347),
          ('MAC000002', 'Std', '2012-11-10', 13.245000000000001),
          ('MAC000002', 'Std', '2012-11-12', 12.321000000000002),
          ('MAC000002', 'Std', '2012-11-13', 10.264000000000001),
          ('MAC000002', 'Std', '2012-11-22', 9.44),
          ('MAC000002', 'Std', '2012-12-08', 13.441999999999998),
          ('MAC000002', 'Std', '2012-12-28', 11.157),
          ('MAC000002', 'Std', '2013-01-01', 10.8),
          ('MAC000002', 'Std', '2013-01-06', 10.293000000000001),
          ('MAC000002', 'Std', '2013-01-11', 10.978999899999996)]
```

And to finish we can convert to a single Pandas dataframe.

```
In [25]: test_summary_daily = pd.DataFrame(
             data = aggregated_test_data_table.collect(),
             columns = ['Household ID', 'Tariff Type', 'Date', 'kWh']
         )
         test_summary_daily
```

Out[25]:

	Household ID	Tariff Type	Date	kWh
0	MAC000002	Std	2012-10-12	7.098
1	MAC000002	Std	2012-10-18	10.751
2	MAC000002	Std	2012-10-24	15.537
3	MAC000002	Std	2012-10-25	13.128
4	MAC000002	Std	2012-11-01	12.209
...	...	...	...	...
205	MAC000002	Std	2013-04-12	12.563
206	MAC000002	Std	2013-04-16	10.940
207	MAC000002	Std	2013-04-25	9.481
208	MAC000002	Std	2013-04-27	9.625
209	MAC000002	Std	2013-05-06	7.418

210 rows × 4 columns

Operating on the full data

Now we can apply the same operations to the full data. Note that nothing will happen until we call `collect`.

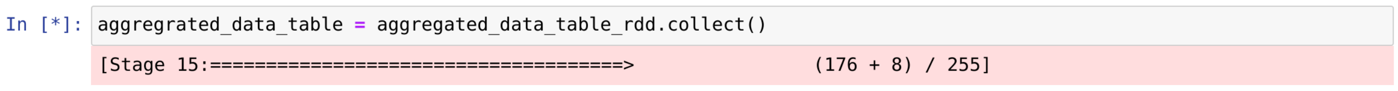
Where we can we amalgamate our functions together into a single function to be called by `map`.

In [26]:
```
def process_data(x):
    x = convert_kwh_data_to_numeric(x)
    x = convert_timestamp_to_date_string(x)
    x = create_keyed_kwh_data(x)
    return x
```

In [27]:
```
aggregated_data_table_rdd = (
    full_data.distinct()
    .map(split_line_into_list)
    .filter(remove_nulls)
    .map(process_data)
    .reduceByKey(lambda x, y : x + y)
    .map(lambda x : (x[0][0], x[0][1], x[0][2], x[1]))
)
```

In [28]:
```
aggregrated_data_table = aggregated_data_table_rdd.collect()
```

Note that when we call `collect` PySpark give us a nice progress bar (example below):



```
In [*]: aggregrated_data_table = aggregated_data_table_rdd.collect()
[Stage 15:=====================================>                (176 + 8) / 255]
```

And now that we have reduced our data we can convert to a single Pandas dataframe.

In [29]:
```
daily_summary = pd.DataFrame(
    data = aggregrated_data_table,
    columns = ['Household ID', 'Tariff Type', 'Date', 'kWh']
)
```

```
In [30]: daily_summary
```

```
Out[30]:
```

	Household ID	Tariff Type	Date	kWh
0	MAC000004	Std	2012-07-22	1.498
1	MAC000007	Std	2013-06-12	6.837
2	MAC000008	Std	2013-04-02	22.690
3	MAC000009	Std	2013-07-06	6.291
4	MAC000012	Std	2013-04-28	1.269
...	...	...	...	...
3510398	MAC002748	Std	2014-02-28	0.115
3510399	MAC003631	Std	2014-02-28	0.068
3510400	MAC005337	Std	2014-02-28	1.325
3510401	MAC002930	ToU	2014-02-28	0.118
3510402	MAC002985	ToU	2014-02-28	0.051

3510403 rows × 4 columns

The rest of this notebook is now essentially the same processing as applied in all the other notebooks in the series.

Saving aggregated data

Now that we have reduced the data down to about 3 million rows it should be managable in a single dataframe. It's useful to save the data so that we don't have to re-run the aggregation every time we want to work on the aggregated data.

We'll save it in a compressed gz format - pandas automatically recognizes the filetype we specify.

```
In [31]: daily_summary.to_csv("data/daily-summary-data.gz", index=False)
```

Analysing the data

```
In [32]: saved_daily_summary = pd.read_csv("data/daily-summary-data.gz")
```

```
In [33]: saved_daily_summary
```

Out[33]:

	Household ID	Tariff Type	Date	kWh
0	MAC000004	Std	2012-07-22	1.498
1	MAC000007	Std	2013-06-12	6.837
2	MAC000008	Std	2013-04-02	22.690
3	MAC000009	Std	2013-07-06	6.291
4	MAC000012	Std	2013-04-28	1.269
...	...	...	...	...
3510398	MAC002748	Std	2014-02-28	0.115
3510399	MAC003631	Std	2014-02-28	0.068
3510400	MAC005337	Std	2014-02-28	1.325
3510401	MAC002930	ToU	2014-02-28	0.118
3510402	MAC002985	ToU	2014-02-28	0.051

3510403 rows × 4 columns

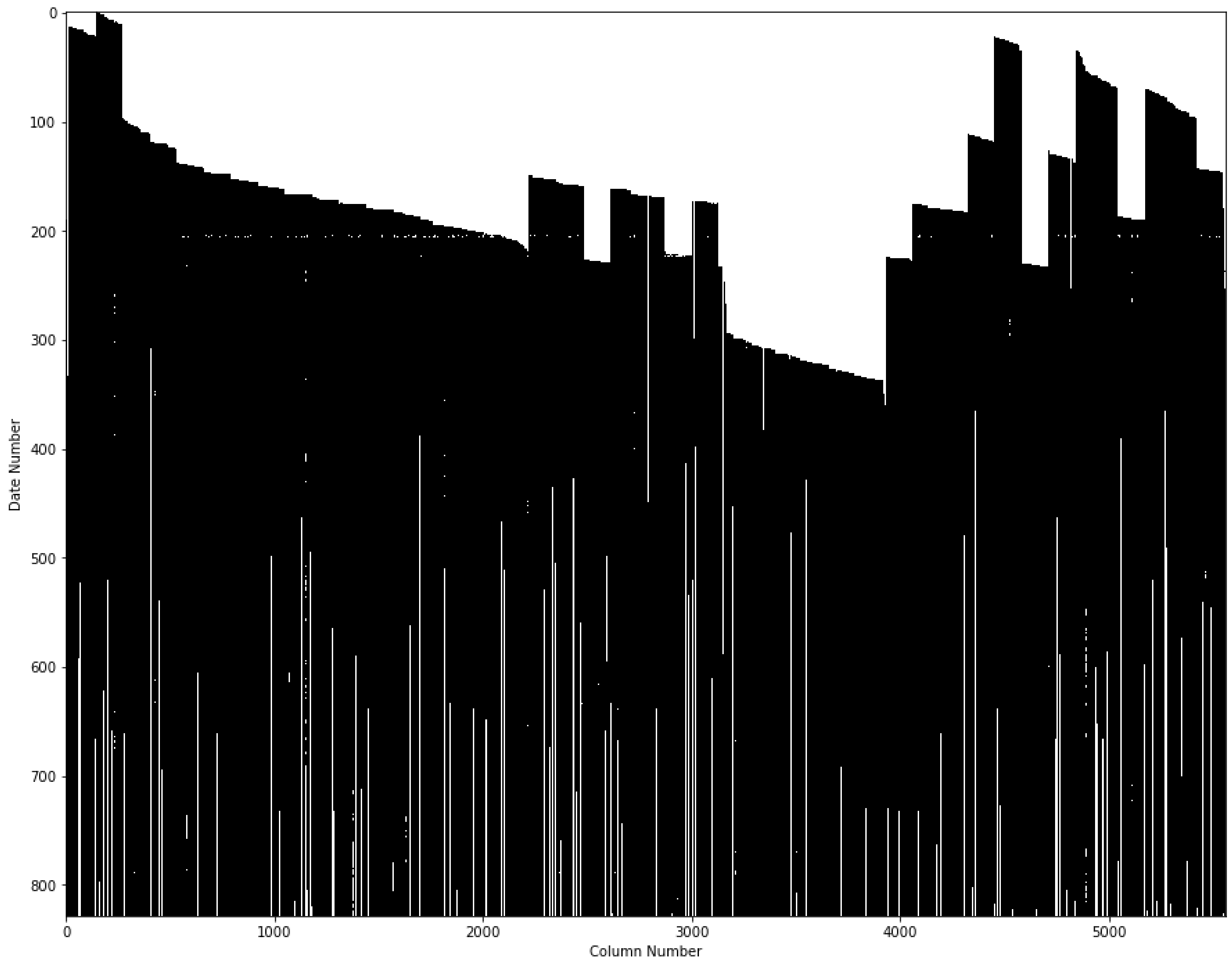
Out of interest let's see what sort of data coverage we have. First we re-organize so that we have households as columns and dates as rows.

In [34]:
```python
summary_table = saved_daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Household ID',
    aggfunc='sum'
)
```

Then we can plot where we have data (black) and where we don't (white).

In [35]:
```python
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 12))
plt.imshow(summary_table.isna(), aspect="auto", interpolation="nearest", cmap="gray")
plt.xlabel("Column Number")
plt.ylabel("Date Number");
```



Despite a slightly patchy data coverage, averaging by tariff type across all households for each day should give us a useful comparison.

In [36]:
```python
daily_mean_by_tariff_type = saved_daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Tariff Type',
    aggfunc='mean'
)
daily_mean_by_tariff_type
```

Out[36]:

Tariff Type	Std	ToU
Date		
2011-11-23	7.430000	4.327500
2011-11-24	8.998333	6.111750
2011-11-25	10.102885	6.886333
2011-11-26	10.706257	7.709500
2011-11-27	11.371486	7.813500
...	...	...
2014-02-24	10.580187	9.759439
2014-02-25	10.453365	9.683862
2014-02-26	10.329026	9.716652
2014-02-27	10.506416	9.776561
2014-02-28	0.218075	0.173949

829 rows × 2 columns

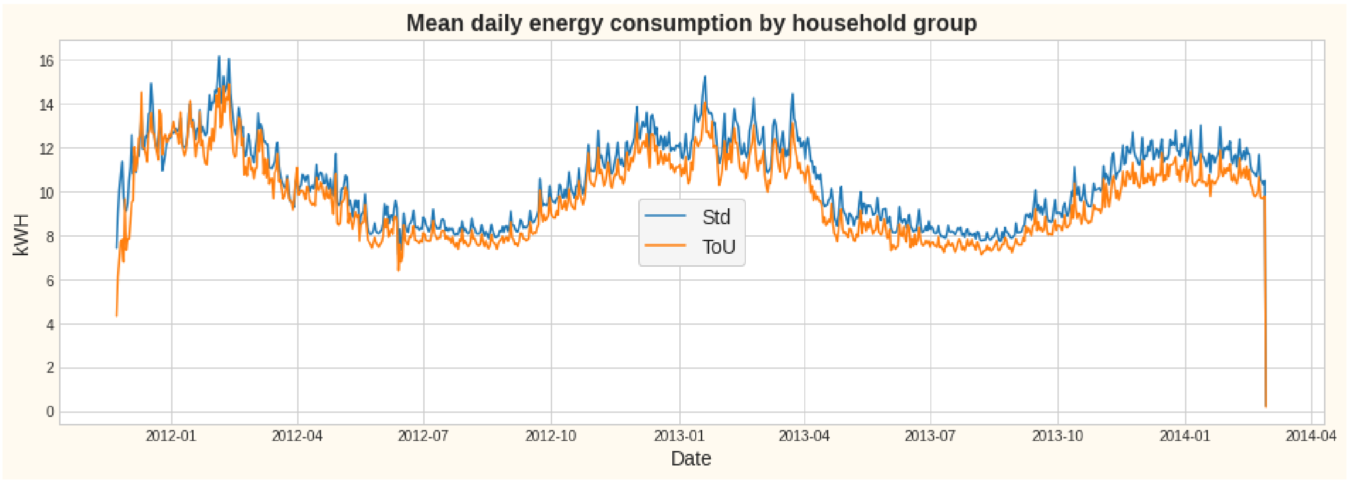
Finally we can plot the two sets of data. The plotting works better if we convert the date from type `string` to type `datetime`.

In [37]:
```
daily_mean_by_tariff_type.index = pd.to_datetime(daily_mean_by_tariff_type.index)
```

In [38]:
```
plt.style.use('seaborn-whitegrid')

plt.figure(figsize=(16, 5), facecolor='floralwhite')
for tariff in daily_mean_by_tariff_type.columns.to_list():
    plt.plot(
        daily_mean_by_tariff_type.index.values,
        daily_mean_by_tariff_type[tariff],
        label = tariff
    )

plt.legend(loc='center', frameon=True, facecolor='whitesmoke', framealpha=1, fontsize=14)
plt.title(
    'Mean daily energy consumption by household group',
    fontdict = {'fontsize' : 16, 'fontweight' : 'bold'}
)
plt.xlabel('Date', fontsize = 14)
plt.ylabel('kWH', fontsize = 14)
plt.show()
```



The pattern looks seasonal which makes sense given heating energy demand.

It also looks like there's a difference between the two groups with the ToU group tending to consume less, but the display is too granular. Let's aggregate again into months.

In [39]:
```
daily_mean_by_tariff_type
```

Out[39]:

Tariff Type	Std	ToU
Date		
2011-11-23	7.430000	4.327500
2011-11-24	8.998333	6.111750
2011-11-25	10.102885	6.886333
2011-11-26	10.706257	7.709500
2011-11-27	11.371486	7.813500
...	...	...
2014-02-24	10.580187	9.759439
2014-02-25	10.453365	9.683862
2014-02-26	10.329026	9.716652
2014-02-27	10.506416	9.776561
2014-02-28	0.218075	0.173949

829 rows × 2 columns

We can see that the data starts partway through November 2011, so we'll start from 1 December. It looks like the data finishes perfectly at the end of February, but the last value looks suspiciously low compared to the others. It seems likely the data finished part way through the last day. This may be a problem elsewhere in the data too, but it shouldn't have an enormous effect as at worst it will reduce the month's energy consumption for that household by two days (one at the beginning and one at the end).

In [40]:
```
monthly_mean_by_tariff_type = daily_mean_by_tariff_type['2011-12-01' : '2014-01-
31'].resample('M').sum()
monthly_mean_by_tariff_type
```

Out[40]:

Tariff Type	Std	ToU
Date		
2011-12-31	377.218580	365.145947
2012-01-31	401.511261	386.016403
2012-02-29	395.065321	368.475150
2012-03-31	349.153085	330.900633
2012-04-30	314.173857	296.903425
2012-05-31	281.666428	263.694338
2012-06-30	257.204029	238.417505
2012-07-31	260.231952	244.641359
2012-08-31	253.939017	238.904096
2012-09-30	266.392972	248.707929
2012-10-31	318.214026	299.714701
2012-11-30	347.818025	326.651435
2012-12-31	390.616106	364.754528
2013-01-31	398.004581	366.548143
2013-02-28	352.189818	325.298845
2013-03-31	381.191994	351.371278
2013-04-30	307.857771	277.856327
2013-05-31	280.762752	256.292247
2013-06-30	254.399013	234.481016
2013-07-31	252.609890	234.104814
2013-08-31	247.046087	231.347310
2013-09-30	267.024791	245.597424
2013-10-31	299.533302	274.332936
2013-11-30	338.082197	306.942424
2013-12-31	369.381371	337.331504
2014-01-31	364.225310	331.578243

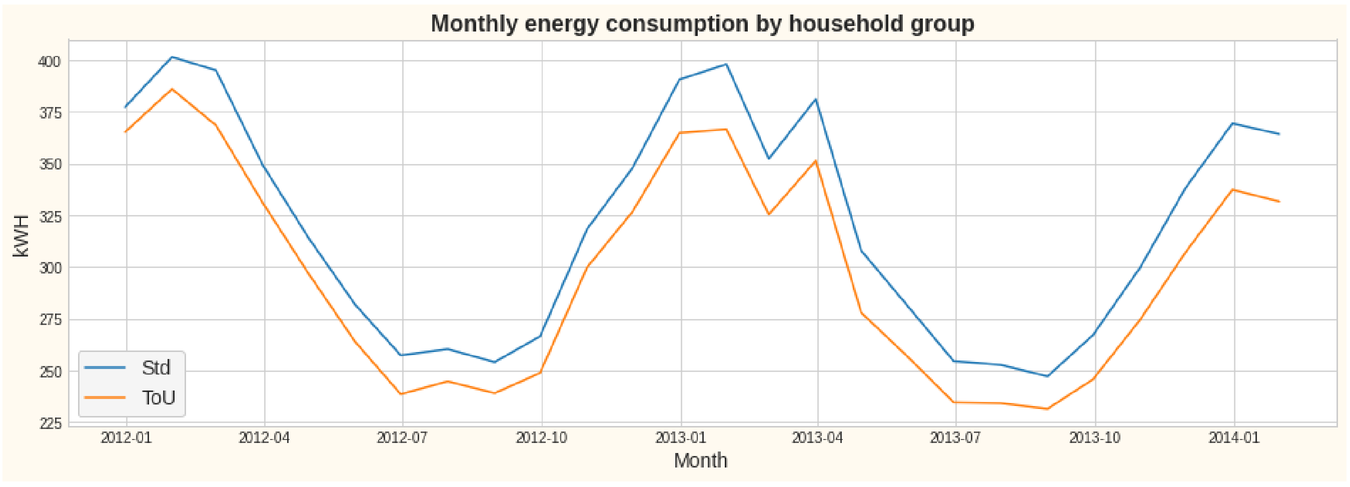
In [41]:

```python
plt.figure(figsize=(16, 5), facecolor='floralwhite')
for tariff in daily_mean_by_tariff_type.columns.to_list():
    plt.plot(
        monthly_mean_by_tariff_type.index.values,
        monthly_mean_by_tariff_type[tariff],
        label = tariff
    )

plt.legend(loc='lower left', frameon=True, facecolor='whitesmoke', framealpha=1, fontsize=14)
plt.title(
    'Monthly energy consumption by household group',
    fontdict = {'fontsize' : 16, 'fontweight' : 'bold'}
)
plt.xlabel('Month', fontsize = 14)
plt.ylabel('kWH', fontsize = 14)
```

```
# Uncomment for a copy to display in results
# plt.savefig(fname='images/result1-no-dupes.png', bbox_inches='tight')

plt.show()
```



The pattern is much clearer and there is an obvious difference between the two groups of consumers.

Note that the chart does not show mean monthly energy consumption, but the sum over each month of the daily means. To calculate true monthly means we would need to drop the daily data for each household where the data was incomplete for a month. Our method should give a reasonable approximation.