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:

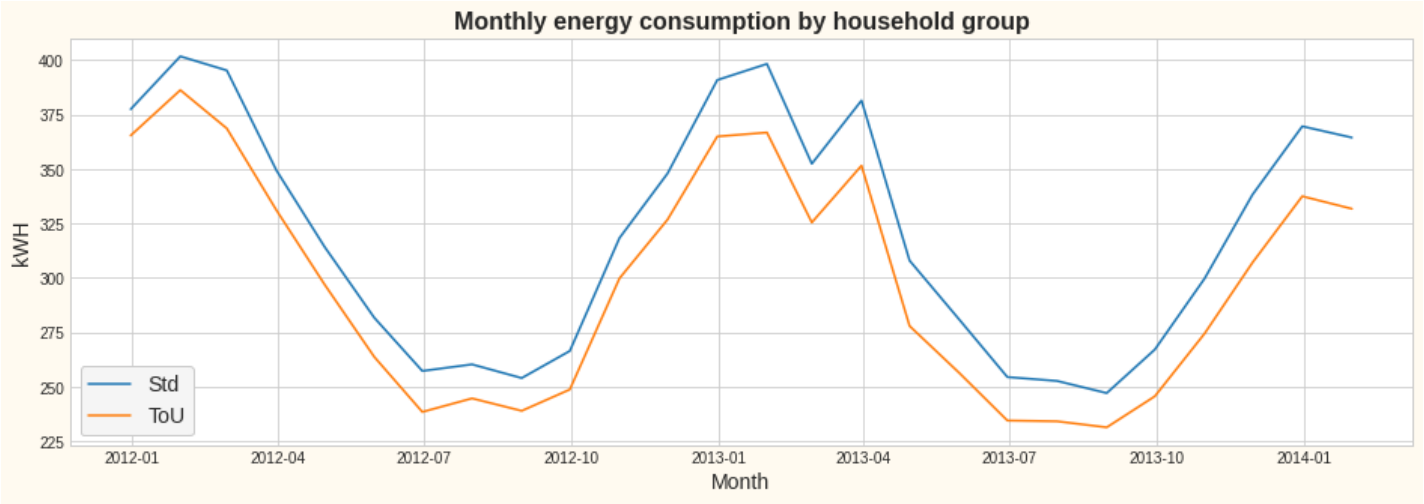
> Energy consumption readings for a sample of 5,567 London Households that took part in the UK Power Networks led Low Carbon London project between November 2011 and February 2014.

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 [1]:
```
!unzip "data/LCL-FullData.zip" -d "data"
```

```
Archive:  data/LCL-FullData.zip
  inflating: data/CC_LCL-FullData.csv
```

Examining the data

In [2]:
```
import findspark
findspark.init()
import pyspark
```

In [3]:
```
import pandas as pd
```

First we must create a `SparkContext`.

In [4]:
```
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).
22/06/01 08:51:25 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
22/06/01 08:51:26 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port
4041.
```

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

In [5]:
```
raw_data = sc.textFile("data/CC_LCL-FullData.csv")
```

In [6]:
```
raw_data.take(15)
```

Out[6]:
```
['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 [7]:
```
raw_data.getNumPartitions()
```

Out[7]:
```
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 [8]:
```
header = raw_data.first()
full_data = raw_data.filter(lambda x : x != header)
full_data.take(15)
```

Out[8]:
```
['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 [9]:
```
test_data = sc.parallelize(full_data.take(10000))
test_data.getNumPartitions()
```

Out[9]:
```
8
```

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 [10]:
```
def split_line_into_list(x):
    '''
    Split single string of line in csv format into values corresponding to columns
    '''
    return x.split(",")
```

In [11]:
```
split_test_data = test_data.map(split_line_into_list)
```

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 [12]:
```
split_test_data.collect()[:15]
```

Out[12]:
```
[['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 ']]
```

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 [13]:
```
def convert_kwh_data_to_numeric(x):
    '''
    Convert the kWh value from string to float
    '''
    x[3] = float(x[3])
    return x
```

In [14]:
```
numeric_kwh_test_data = split_test_data.map(convert_kwh_data_to_numeric)
numeric_kwh_test_data.collect()
```

```
22/06/01 08:51:29 ERROR Executor: Exception in task 3.0 in stage 5.0 (TID 15)
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_8700/95641023.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 [15]:
```
def remove_nulls(x):
    return x[3] != 'Null'
```

In [16]:
```
filtered_test_data = split_test_data.filter(remove_nulls)
```

In [17]:
```
numeric_kwh_test_data = filtered_test_data.map(convert_kwh_data_to_numeric)
numeric_kwh_test_data.collect()[:15]
```

Out[17]:
```
[['MAC000002', 'Std', '2012-10-12 00:30:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 01:00:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 01:30:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 02:00:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 02:30:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 03:00:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 03:30:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 04:00:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 04:30:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 05:00:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 05:30:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 06:00:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 06:30:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 07:00:00.0000000', 0.0],
 ['MAC000002', 'Std', '2012-10-12 07:30:00.0000000', 0.0]]
```

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

In [18]:
```
def convert_timestamp_to_date_string(x):
    '''
    Convert the timestamp into a date string
    '''
    x[2] = x[2].split(" ")[0]
    return x
```

In [19]:
```
cleaned_test_data = numeric_kwh_test_data.map(convert_timestamp_to_date_string)
cleaned_test_data.collect()[:15]
```

Out[19]:
```
[['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0],
 ['MAC000002', 'Std', '2012-10-12', 0.0]]
```

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 [20]:
```
def create_keyed_kwh_data(x):
    return ( (x[0], x[1], x[2]), x[3] )
```

In [21]:
```
keyed_kwh_data = cleaned_test_data.map(create_keyed_kwh_data)
keyed_kwh_data.collect()[:15]
```

Out[21]:
```
[(('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0),
 (('MAC000002', 'Std', '2012-10-12'), 0.0)]
```

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

In [22]:
```
aggregated_test_data = keyed_kwh_data.reduceByKey(lambda x, y : x + y)
aggregated_test_data.collect()[:15]
```

Out[22]:
```
[(('MAC000002', 'Std', '2012-10-12'), 7.098000000000001),
 (('MAC000002', 'Std', '2012-10-18'), 10.751),
 (('MAC000002', 'Std', '2012-10-24'), 15.537000099999997),
 (('MAC000002', 'Std', '2012-10-25'), 13.128),
 (('MAC000002', 'Std', '2012-11-01'), 12.208999999999998),
 (('MAC000002', 'Std', '2012-11-03'), 14.347000000000001),
 (('MAC000002', 'Std', '2012-11-10'), 13.245000000000003),
 (('MAC000002', 'Std', '2012-11-12'), 12.321),
 (('MAC000002', 'Std', '2012-11-13'), 10.264),
 (('MAC000002', 'Std', '2012-11-22'), 9.439999999999998),
 (('MAC000002', 'Std', '2012-12-08'), 13.442),
 (('MAC000002', 'Std', '2012-12-28'), 11.157000000000002),
 (('MAC000002', 'Std', '2013-01-01'), 10.799999999999997),
 (('MAC000002', 'Std', '2013-01-06'), 10.293000000000003),
 (('MAC000002', 'Std', '2013-01-11'), 10.9789999)]
```

In [23]:
```
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[23]:
```
[('MAC000002', 'Std', '2012-10-12', 7.098000000000001),
 ('MAC000002', 'Std', '2012-10-18', 10.751),
 ('MAC000002', 'Std', '2012-10-24', 15.537000099999997),
 ('MAC000002', 'Std', '2012-10-25', 13.128),
 ('MAC000002', 'Std', '2012-11-01', 12.208999999999998),
 ('MAC000002', 'Std', '2012-11-03', 14.347000000000001),
 ('MAC000002', 'Std', '2012-11-10', 13.245000000000003),
 ('MAC000002', 'Std', '2012-11-12', 12.321),
 ('MAC000002', 'Std', '2012-11-13', 10.264),
 ('MAC000002', 'Std', '2012-11-22', 9.439999999999998),
 ('MAC000002', 'Std', '2012-12-08', 13.442),
 ('MAC000002', 'Std', '2012-12-28', 11.157000000000002),
 ('MAC000002', 'Std', '2013-01-01', 10.799999999999997),
 ('MAC000002', 'Std', '2013-01-06', 10.293000000000003),
 ('MAC000002', 'Std', '2013-01-11', 10.9789999)]
```

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

In [24]:
```
test_summary_daily = pd.DataFrame(
    data = aggregated_test_data_table.collect(),
    columns = ['Household ID', 'Tariff Type', 'Date', 'kWh']
)
test_summary_daily
```

Out[24]:

	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 [25]:
```
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 [26]:
```
aggregated_data_table_rdd = (
    full_data.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 [27]:
```
aggregated_data_table = aggregated_data_table_rdd.collect()
```

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

In [*]:
```
aggregated_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 [28]:
```
daily_summary = pd.DataFrame(
    data = aggregated_data_table,
    columns = ['Household ID', 'Tariff Type', 'Date', 'kWh']
)
```

In [29]:
```
daily_summary
```

Out[29]:

	Household ID	Tariff Type	Date	kWh
0	MAC000002	Std	2012-11-11	10.699
1	MAC000002	Std	2013-05-13	8.240
2	MAC000003	Std	2012-07-24	14.177
3	MAC000003	Std	2013-06-12	10.921
4	MAC000004	Std	2012-06-25	2.115
...	...	...	...	...
3510398	MAC005550	ToU	2014-02-17	10.892
3510399	MAC005551	ToU	2014-01-08	8.586
3510400	MAC005557	ToU	2013-01-25	6.514
3510401	MAC005564	ToU	2012-09-17	5.431
3510402	MAC005564	ToU	2012-12-16	3.497

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 [30]:
```
daily_summary.to_csv("data/daily-summary-data.gz", index=False)
```

Analysing the data

In [31]:
```
saved_daily_summary = pd.read_csv("data/daily-summary-data.gz")
```

In [32]:
```
saved_daily_summary
```

Out[32]:

	Household ID	Tariff Type	Date	kWh
0	MAC000002	Std	2012-11-11	10.699
1	MAC000002	Std	2013-05-13	8.240
2	MAC000003	Std	2012-07-24	14.177
3	MAC000003	Std	2013-06-12	10.921
4	MAC000004	Std	2012-06-25	2.115
...	...	...	...	...
3510398	MAC005550	ToU	2014-02-17	10.892
3510399	MAC005551	ToU	2014-01-08	8.586
3510400	MAC005557	ToU	2013-01-25	6.514
3510401	MAC005564	ToU	2012-09-17	5.431
3510402	MAC005564	ToU	2012-12-16	3.497

3510403 rows × 4 columns

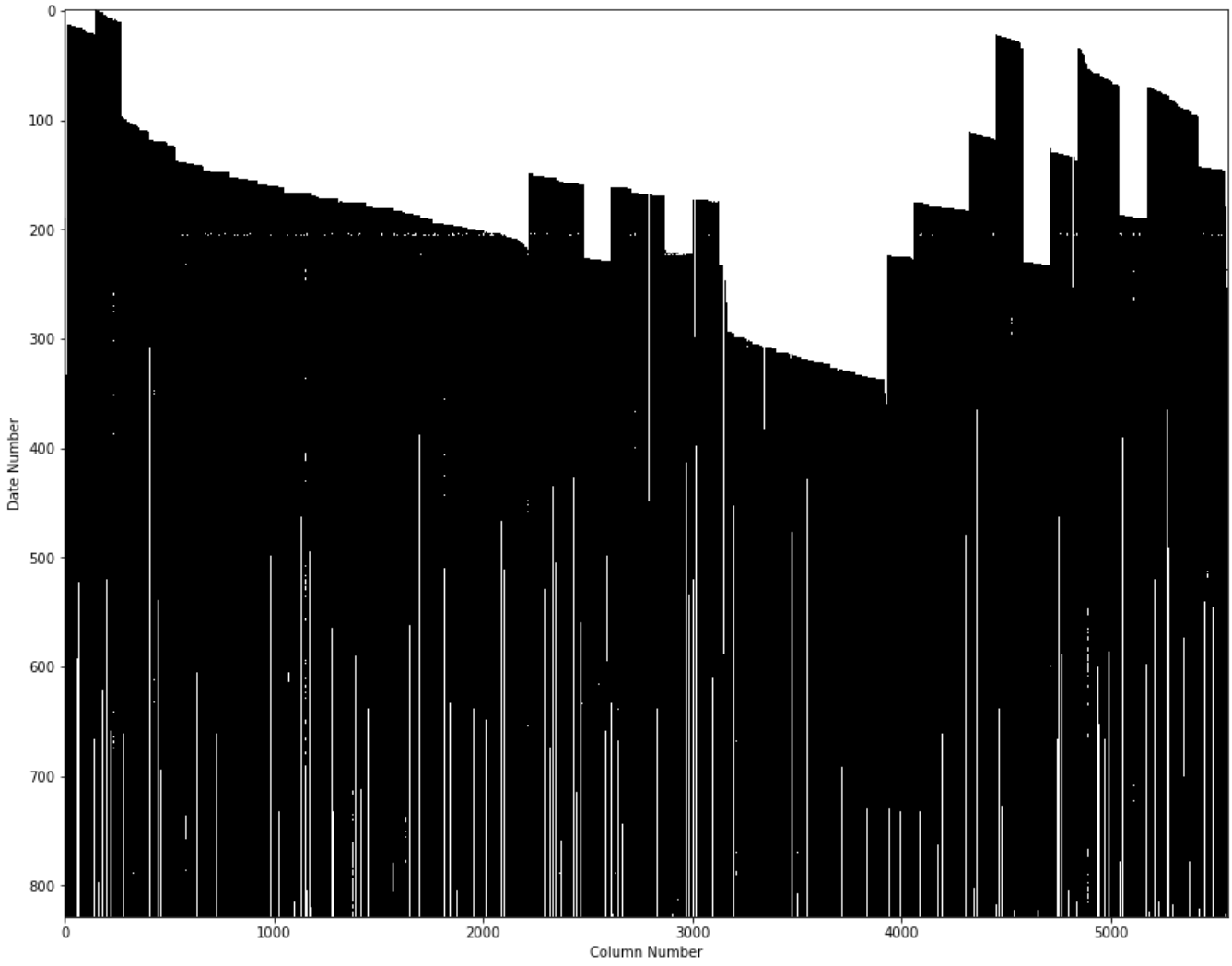
Out of interest let's see what sort of data coverage we have. First we re-organize so that we have households as columns and dates as rows.

In [33]:
```
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 [34]:
```
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 [35]:
```
daily_mean_by_tariff_type = saved_daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Tariff Type',
    aggfunc='mean'
)
daily_mean_by_tariff_type
```

Out[35]:

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	0.436149	0.347899

829 rows × 2 columns

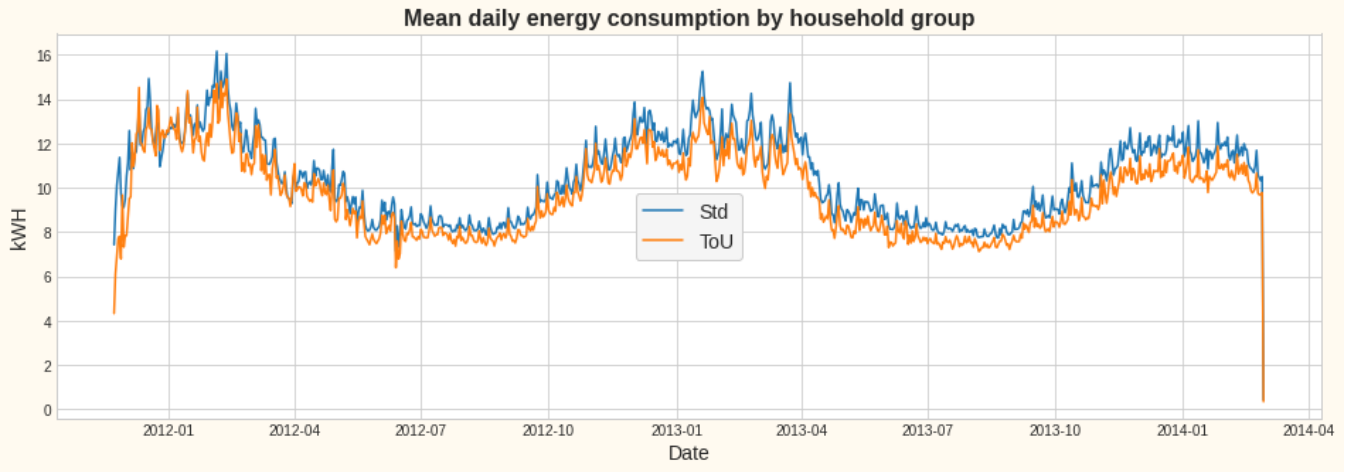
Finally we can plot the two sets of data. The plotting works better if we convert the date from type `string` to type `datetime`.

In [36]:
```
daily_mean_by_tariff_type.index = pd.to_datetime(daily_mean_by_tariff_type.index)
```

In [37]:
```
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 [38]:
```
daily_mean_by_tariff_type
```

Out[38]:

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	0.436149	0.347899

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 [39]:
```
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[39]:

Tariff Type	Std	ToU
Date		
2011-12-31	377.443042	365.391597
2012-01-31	401.744672	386.253703
2012-02-29	395.294296	368.663764
2012-03-31	349.367317	331.095386
2012-04-30	314.323216	297.032370
2012-05-31	281.796440	263.812879
2012-06-30	257.333248	238.532452
2012-07-31	260.359313	244.757999
2012-08-31	254.085724	239.041805
2012-09-30	266.515247	248.820055
2012-10-31	318.361735	299.849633
2012-11-30	348.007365	326.831890
2012-12-31	390.864676	364.969958
2013-01-31	398.275908	366.779573
2013-02-28	352.440444	325.489548
2013-03-31	381.472409	351.591760
2013-04-30	308.005098	277.976132
2013-05-31	280.934227	256.428977
2013-06-30	254.542531	234.591000
2013-07-31	252.761147	234.224724
2013-08-31	247.190593	231.464453
2013-09-30	267.165424	245.707678
2013-10-31	299.703934	274.464114
2013-11-30	338.317167	307.131828
2013-12-31	369.630558	337.524715
2014-01-31	364.460042	331.767440

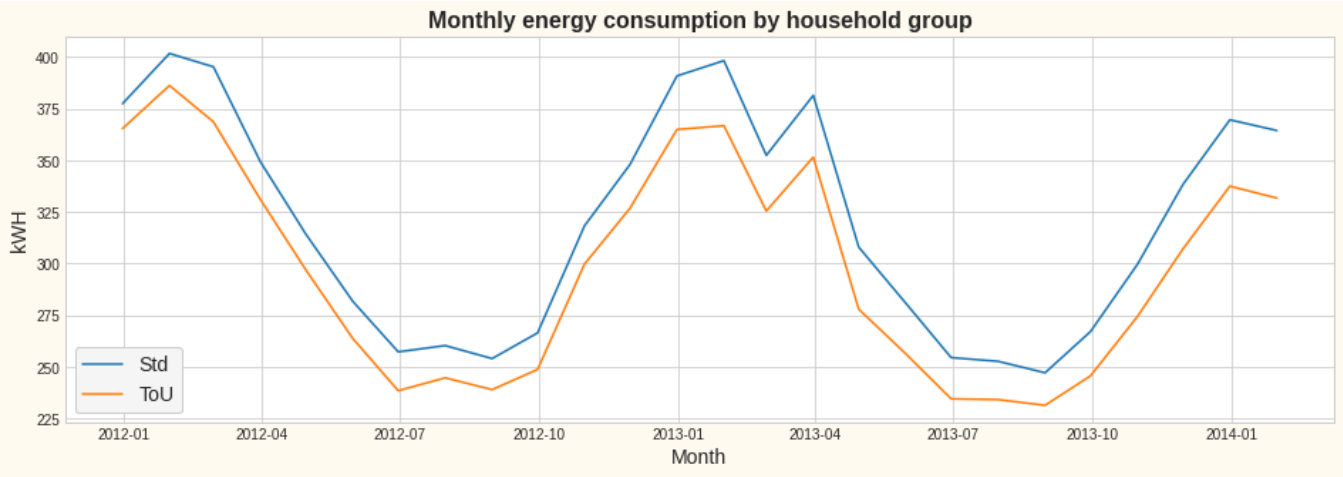
In [40]:

```
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.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.