A Scalable Smart Meter Data Generator Using Spark

Iftikhar, Nadeem; Liu, Xiufeng; Danalachi, Sergiu; Nordbjerg, Finn; Vollesen, Jens

Published in:
OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"

Link to article, DOI:
10.1007/978-3-319-69462-7_2

Publication date:
2017

Document Version
Peer reviewed version

A Scalable Smart Meter Data Generator Using Spark

Nadeem Iftikhar[1], Xiufeng Liu[2], Sergiu Danalachi[1], Finn Ebertsen Nordbjerg[1] and Jens Henrik Vollesen[1]

[1] University College of Northern Denmark
{naif,1028752,fen,hnv}@ucn.dk
[2] Technical University of Denmark
xiuli@dtu.dk

Abstract. Today, smart meters are being used worldwide. As a matter of fact smart meters produce large volumes of data. Thus, it is important for smart meter data management and analytics systems to process petabytes of data. Benchmarking and testing of these systems require scalable data, however, it can be challenging to get large data sets due to privacy and/or data protection regulations. This paper presents a scalable smart meter data generator using Spark that can generate realistic data sets. The proposed data generator is based on a supervised machine learning method that can generate data of any size by using small data sets as seed. Moreover, the generator can preserve the characteristics of data with respect to consumption patterns and user groups. This paper evaluates the proposed data generator in a cluster based environment in order to validate its effectiveness and scalability.

Keywords: Smart meter, scalable, synthetic data generator, time-series

1 Introduction

Today, with the popularity of Internet of Things (IoT) and cloud computing, the size of data grows exponentially, posing new challenges to data analysis and management systems, such as the ability to handle petabytes of data. Traditionally, simple benchmarks have been largely used for evaluating the systems in order to prevent unnecessary complexity. On the other hand, we believe that benchmarking should meet a certain diversity and workload requirement for obtaining meaningful results. In addition, it is preferable to use realistic data, however, it is quite challenging to obtain a considerable size of domain dependent data for benchmarking and experimentation purposes. For example, limited public data sets are available in the energy sector. Often, it is difficult to obtain a truthful data source, primarily due to data privacy laws or high data storage cost. Storing petabytes of data is still fairly expensive, although it is much cheaper than before. For example, one TB standard hard drive costs about $\$80$, approximately $\$0.08$ per GB. Similarly, the price for one PB of disk space approximately costs about $\$80,000$. Hence, it is meaningless to store petabyte data only for testing purpose. In addition to data storage, it is also costly to transport large amounts of data over the network, which may consume bandwidth and time. For that reason, scalable data should be produced and used as needed.

In the energy sector, smart meter data management and analysis have received considerable effort in recent years, due to the widespread deployment of smart meters. A smart meter reads energy consumption at a regular time interval, typically every 15 minutes and sends readings back to an energy data management system for monitoring and billing purposes [1]. Thus, it is essential to evaluate the performance, robustness of energy data management systems and to investigate suitable technologies and algorithms for smart meter data analytics [2–4]. In order to test these systems, it is feasible to generate scalable data sets that should reflect the characteristics of real-world energy consumption patterns. For example, residential energy consumption usually follows a regular pattern based on the consumption habits of a household. Figure 1, illustrates a typical weekly electricity consumption time series from Irish open data [5]. It can be observed that this household have roughly a fixed consumption pattern. The time series has a morning peak roughly at 7-8 o'clock during the workdays. Further, the morning peak delays to around 10 o'clock during the weekend. In the evening, there is a considerable evening peak between 18:00 and 23:00, when all the family members are home and the electric appliances might be turned on, such as dish washer, cooking range, washing machine, television and so on.

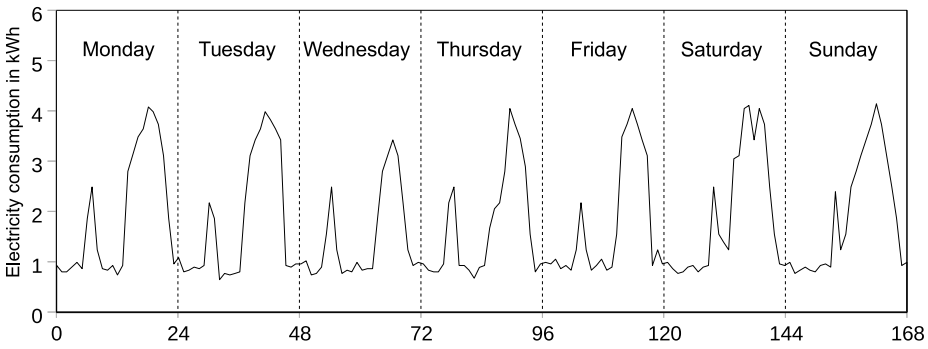

Fig. 1. Weekly consumption pattern of a typical private household

In this paper, we present a scalable data generator that can generate huge volume of realistic synthetic data. The data generator takes as input a real-world energy consumption time series as seed and generates synthetic time series based on historical consumption patterns. In doing so, the generator first creates an adjusted time series using a moving average time series model. The moving average reduces the periodic variations from the actual time series by smoothing the peak periods. Then, it uses autoregressive time series model to predict meter readings. In the end, the periodic variations are added back to the newly predicted meter readings to reflect the pattern and variance of the real-world energy consumption. The data generator is implemented by using the memory-based distributed computing framework, Spark, which can generate scalable data sets on a cluster based environment.

This paper is a significant extension of the previous work [6]. In the previous work, the concept of prediction-based smart meter data generation was introduced, however, it remains to prove that the single machine based data processing platform introduced in [6] also works for cluster-based platform. A scalable data generator is the next step. In t his paper, the single machine based technique is extended by introducing the cluster-based technique.

Our main contributions in this paper are as follows:

- We propose a scalable smart meter data generator using Spark.
- We propose a novel method of generating realistic data sets that can preserve the characteristics of real-world energy consumption time series, including patterns and user-groups.
- We evaluate the data generator in terms of effectiveness and scalability of generating scalable data sets, with relatively small data as seed.

The paper is structured as follows. Section 2 describes the methodology used by the proposed data generator. Section 3 describes the implementation on Spark. Section 4 evaluates the generator. Section 5 presents the related work. Section 6 concludes the paper and points out the future research directions.

2 Methodology

2.1 Overview

We now describe the rationale of the proposed data generation solution. The solution uses a *quantitative model*, expressed in mathematical notation. The quantitative model is further divided into a causal model and a time-series model, where the latter is chosen for modeling the consumption time series. The time series model produces predictions according to historical consumption patterns. The time series of residential energy consumption normally comprises the following patterns: *trend*, *cyclic* and *seasonal/periodic*. The periodic pattern is usually resulted from the periodical factors such as the days, which have a fixed and known period [7], e.g., 24-hour. Therefore, it is possible to generate consumption time series with these pattern characteristics.

Figure 2, gives the overview of the data generation process. The data generation is seeded by a small real-world data set. First, the seed data is deseasonalized in order to flatten the periodic variations. Next, a regression model is trained using the flattened time series. This model is then used to predict new consumption values. In the end, the generated time series is reseasonalized, in other words, the periodic variations are added back. The rationale of using the adjusted periodic variations is that the data that does not have or has reduced periodic variations can lead to more accurate predictions than with variations [8]. The time series with reduced periodic variations also allows us to determine the best regression model for the prediction.

2.2 Algorithm Description

We now describe the data generation process and the algorithms used. The data generation process comprises of two methods: training process and generation process.

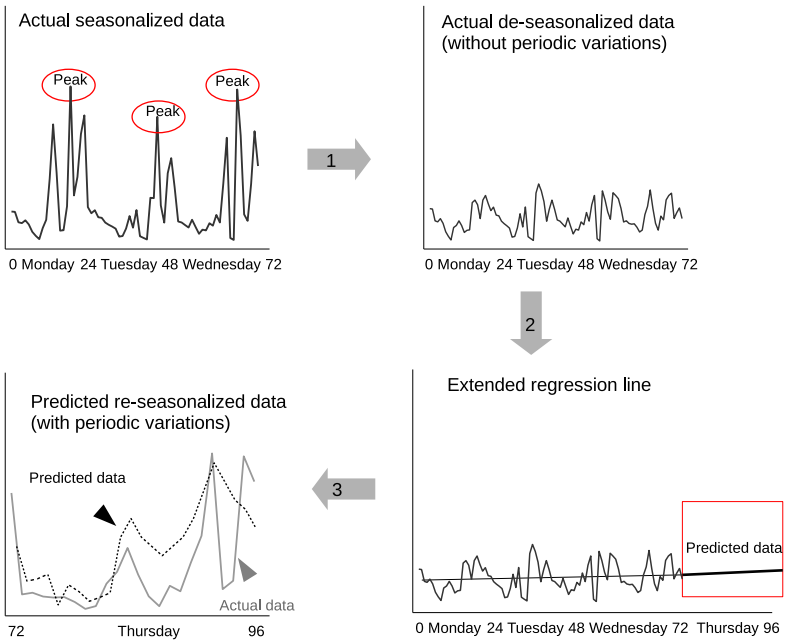

Fig. 2. Data generation overview [6]

The training process includes flattening of time series fluctuations, deseasonalization and generation of data models, while generation process includes generating data using the model and reseasonalization. Both of the processes are described in the following subsections.

Training process. For the proposed data generator, we consider generating data based on daily consumption profiles. During the training process (see Algorithm 1), each time series from the seed data set will be transformed into a key-value pair, of which *meterID* is the key, and the list of meter readings is the value. The readings in the list are sorted in an ascending order according to the timestamps.

Algorithm 1 Training process

1. Transform a time series into a *key-value* pair.
2. Process the *key-value* pair:
   (i) Flatten fluctuations by *centered moving averaging* method.
   (ii) Deseasonalize time series by *periodic indexing* method.
   (iii) Train *autoregressive* model for predictions using the deseasonalized *time series*.
   (iv) Write the models and output: *meterIDs*, *periodic-indices*, *AR-coefficients* and *flatten-time-series*.

Next, the key-value pair is processed through the following four steps (Algorithm 1) that include flattening of periodic fluctuations, deseasonalization, autoregression and writing the output:
(i) Flatten periodic fluctuations: We use the *centered moving averaging (CMA)* method to reduce the impact of periodic fluctuations [9]. CMA replaces the original time series with a new flatten time series where each point is centered at the middle of the data values being averaged. For the daily profile (24-hour), the CMA of an even period is defined as:

$$\begin{aligned}\mathcal{A}(i) = &\tfrac{1}{2}\left(\frac{y_{i-12} + .. + y_i + .. + y_{i+11}}{24}\right) + \\ &\tfrac{1}{2}\left(\frac{y_{i-11} + .. + y_i + .. + y_{i+12}}{24}\right)\end{aligned} \tag{1}$$

where y_i is the i-th observation in a time series of the seed data set.
(ii) Deseasonalization: To deseasonalize a time series, we first need to compute the *raw-index or Ratio-to-Moving-Average*, which is computed as bellow:

$$\mathcal{R}(i) = \frac{y_i}{\mathcal{A}(i)} \tag{2}$$

We then compute the *periodic indices* by using the resulting raw index values (see Equation 3). For each hour of the day, a corresponding periodic index is computed, which is the mean value of all the raw index values at that particular hour. For example, $\mathcal{P}(0)$ represents the mean of all $\mathcal{R}$ values at 0 o'clock in all days for a given time series. Therefore, the total number of resulting periodic indices will be 24.

$$\mathcal{P}\left(h\right) = \frac{1}{n}\sum_{i=0}^{n-1}\mathcal{R}(h + 24i) \tag{3}$$

where, n represents the total number of days for each meter in the time series, and h is the hour of the day, i.e., $0 - 23$.

Since there are some chances to encounter data precision problems, e.g, due to the floating point, we need to adjust the computed $\mathcal{P}$ value [10]. Equation 4 normalizes the periodic indices, which ensures that the sum of the adjusted $\mathcal{P}'$ values is 1.0.

$$\mathcal{P}'(h) = \frac{24 * \mathcal{P}(h)}{\sum_{h=0}^{23}\mathcal{P}(h)} \tag{4}$$

In the end, we use this adjusted periodic indice to deseasonalize a time series, which simply divides each data point of the time series (see Equation 5).

$$y'_i = \frac{y_i}{\mathcal{P}'(h)} \tag{5}$$

where $h = i \; mod \; 24$ and $\mathcal{P}'$ is the normalized periodic indices.
(iii) Training autogressive model and (iv) writing output. In the end, we use the flatten (deseasonlized) time series to train an autoregressive (AR) model and this model will

Algorithm 2 Data generation

1. Read the data from the two input files, and create Spark RDD tables: $\mathcal{PI}$*=(meterID, peridoc-indices)* and $\mathcal{AR}$*=(meterID, AR-coefficients, flatten-time-series)*.
2. Do the *Theta join* on table $\mathcal{PI}$ and $\mathcal{AR}$ where $\mathcal{PI}.meterID \neq \mathcal{AR}.meterID$.
3. For-each query results:
   (i) Predict a new reading by using the *AR* model and the flatten time series values.
   (ii) Reseasonalize the new reading.
   (iii) Add *base load* and *white noise* to the reseasonalised reading in order to simulate reality.

be used to generate new values by prediction. The resulting coefficients of AR model, the periodic indices and the flatten-time-series, $\{y'_i | i = 0, ..., n-1\}$, will be written to the Hadoop distributed file system (HDFS). The results are stored into two separate files, with the formats of *(meterID, periodic-indices)* and *(meterID, (AR-coefficients, flatten-time-series))*. In the data generation process, these two files will be served as input.

Generation process. Algorithm 2, describes the data generation process. The data generator uses the files (generated from the preprocessing process) as input. The data from the two files are read as two RDD tables, $\mathcal{PI}$ and $\mathcal{AR}$, in Spark, and the *theta join* [11] will apply on the two tables on the condition that the meterIDs are not equal. For each record of the join results, we apply the following three steps to generate a new time series:

(i) Generate new reading: We use the AR model and the values from flatten time series (with the order of p) to generate a new value, which is expressed in the following equation:

$$y''_i = c + \sum_{\lambda=1}^{p} \alpha_i y'_{i-\lambda} \tag{6}$$

where c is the intercept with the $y - axis$ (a constant), α is the AR coefficient and y'_i are the last values from the flatten time series of (with p consecutive values before i).
(ii) Reseasonalization and (iii) add base load and white noise: The final resulting time series is

$$y'''_i = y''_i * \mathcal{P}'(h) + baseLoad + \epsilon_i \tag{7}$$

where $h = i \; mod \; 24$ and $i = 0, ..., n$.

The reseasonalization is simply multiplying the adjusted periodic index. In the generated time series, we add a base load, which is a constant value greater or equal to zero. A base load typically represents the energy consumed by the appliance that is always on, e.g., refrigerator. And, we add a Gaussian white noise, $\epsilon \sim \mathcal{N}(0, 1.0)$, to simulate slight variations.

2.3 Optimization

We now optimize our data generator in order to better simulate the real-world data. As mentioned in Section 1, energy consumption data follows a certain pattern, due to

the daily routine of a household, e.g., having a daily pattern with morning and evening peaks. Moreover, the time series of different households may have similar patterns, which can be identified by grouping/clustering. This technique is often used by utilities to segment the customers in order to offer personalized energy-efficiency services. In order not to lose this information, we optimize data generation by adding the pre-processing process (see Figure 3). The pre-processing will first cluster the seed, then uses the clustered data for training the models. Recall that in the data generation process, we use the theta join on the resulting models to create data generators. If the models were not generated by the clustered seed, the resulting synthetic data may lose the clustering information.

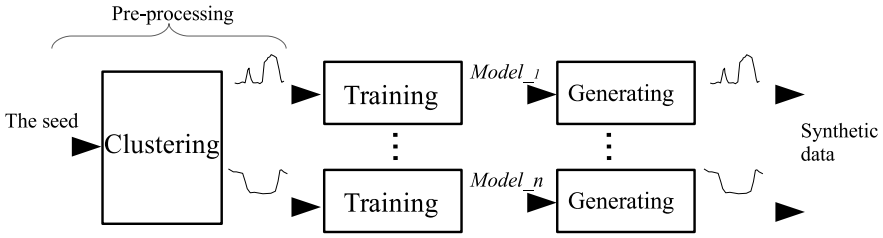

Fig. 3. Optimize data generation with the pre-processing of the seed

Moreover, clustering the seed time series according to daily patterns is a two step process: First, we find the typical daily load pattern for each time series, which is done by averaging the consumption of each hour for all days. This results the following averaging load of daily profile for the i-th time series:

$$\mathcal{TS}_i = \{r_{i,0}, r_{i,1}, .., r_{i,23}\} \tag{8}$$

where r represents the average consumption of a meter at each hour of the day, h.

Second, we cluster the daily load patterns of all time series using *k*-means clustering algorithm [12]. In general, k-means clustering algorithm uses Euclidean distance, e.g., [13, 14], which is defined as follow. Suppose there are two daily load profiles of $\mathcal{TS}_i$ and $\mathcal{TS}_j$, the distance is

$$euclDist\,(\mathcal{TS}_i, \mathcal{TS}_j) = \sqrt{\sum_{h=0}^{23} (r_{i,h} - r_{j,h})^2} \tag{9}$$

However, using the Euclidean distance may still not the best to reflect similarity of two load patterns. For example, Figure 4 (a) and (b) both have the Euclidean distance of $\sqrt{3}$, however, the patterns in Figure 4 (b) are totally different.

To further optimize, we adopt the Pearson correlation distance [15], which measures the distance based on the correlation of between two patterns. The correlation is defined

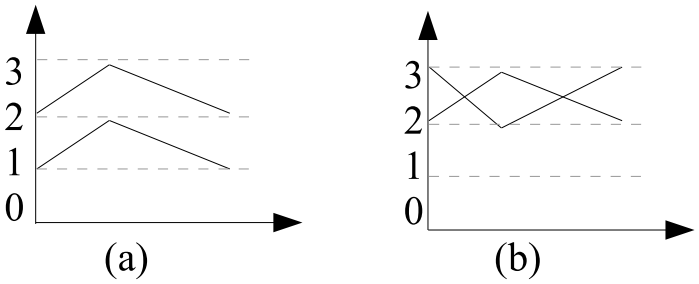

Fig. 4. The two patterns with the same Euclidean distance of $\sqrt{3}$

as follow:

$$corr\left(\mathcal{TS}_i, \mathcal{TS}_j\right) = \frac{\sum_{h=0}^{23}\left(r_{i,h} - \mu_i\right)\left(r_{j,h} - \mu_j\right)}{\sqrt{\sum_{h=0}^{23}\left(r_{i,h} - \mu_i\right)^2}\sqrt{\sum_{h=0}^{23}\left(r_{j,h} - \mu_j\right)^2}} \tag{10}$$

where μ represents the daily average consumption for each meter.

The correlation distance is defined as:

$$corrDist\left(\mathcal{TS}_i, \mathcal{TS}_j\right) = 1 - corr\left(\mathcal{TS}_i, \mathcal{TS}_j\right) \tag{11}$$

The distance of zero represents perfectly correlated (correlation = 1) time series. The distance of less than approximately 0.5 indicates that there is a good similarity between two patterns, while the distance of 2 (correlation = -1) indicates having an opposite pattern.

3 Implementation on Spark

The proposed data generator is implemented into two modules, training module and data generation module, which are both implemented using *Spark* for generating scalable data. The implementations are described as follows.


```
train(inputPath, outputPath, frequency){
    seed = getReadingsForEachMeterID(inputPath).cache()
    output = seed.mapValues(readings => {
          PI = getPeriodicIndices(readings, frequency)
          DS = getDeseasonalizedSeed(readings, PI)
          coefficients = ARIMA.fitModel(3, 0, 0, DS, true).
    coefficients
          lagged = Vectors.dense(DS.takeRight(3))
          (PI, coefficients, lagged)})
    .filter(!coefficients(0).isNaN)
   output.map(tuple => (meterID, PI)).save(outputPath+"/PI")
   output.map(tuple => (meterID, (coefficients, lagged))).save(
    outputPath+"/AR")}
```

Listing 1.1. The code snippet of training

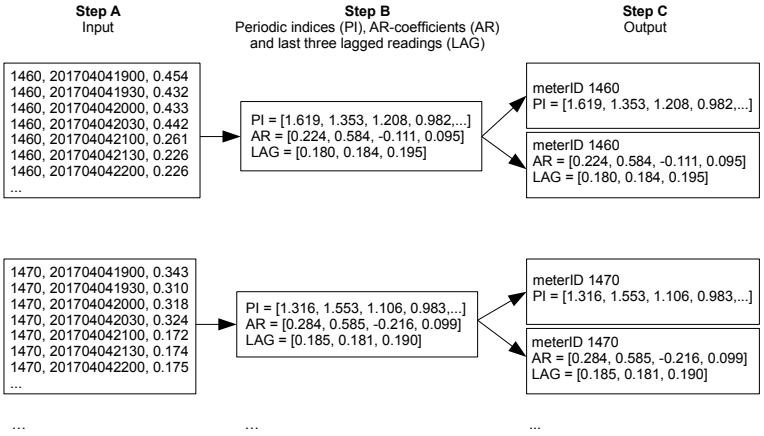

Fig. 5. Training module

The seed data have been processed by grouping/clustering. The training process will take a clustered seed data as the input to create the models. Listing 1.1 shows the code snippet of training process, which takes the parameters of *inputPath*, *outputPath* and *frequency* (line 1). The input path locates a clustered seed data that comprise a set of time series with similar daily consumption patterns. The output path denotes the location of saving the resulting models in HDFS and the frequency indicates the number of occurrences of a meter reading per unit time. For example, frequency=48 represents the reading frequency per day, since the meter is read every 30 minutes. The input files are the CSV files with the format of *(meterID, timestamp, reading)*, where meterID is taken as the *key* and (timestamp, reading) is taken as the *value*. The function on line 2 will sort and groups the readings based on meter id and time and cache the data in memory cache iterative processing. Second, the periodic indices are computed for each time series, so does the deseasonalization (lines 4-5). Third, the AR model is trained (by using the spark-timeseries library) using the deseasonalized time series (line 6). Fourth, three deseasonalized lagged (past period) readings are extracted (with order=3), which will be used for forecasting the new value in the data generation process (line 7). Fifth, the results are mapped as periodic indices, coefficients and lagged readings (line 8). Sixth, the results with undefined coefficients are filtered out (line 9). Last, the results are stored to HDFS directly (lines 10-11). The training process is run only once for each clustered data set from the seed. The two resulting files have the following format: *<meter identifier, periodic indices>* and *<meter identifier, AR-coefficients, flatten-time-series>*. An example of the rows are *<1460, 1.619, 1.353, 1.208, 0.982, .., 1.776>* and *<1460, 0.224, 0.584, -0.111, 0.095, 0.180, 0.184, 0.195>*. The first row represents that a meter (with meterID = 1460) has 48 periodic indices (as the number of occurrences of a meter reading is per half-hour). The second row represents that the meter (with meterID=1406) has an intercept, three AR coefficients (with order=3), and last three lagged readings of the deseasonalized seed data set.

Moreover, Figure 5 depicts an example of the training module. Step A shows the input CSV files that contain clustered seed data. Step B shows the computed periodic indices, AR-coefficients and the extracted last three deseasonalized lagged reading for each time series. Finally (Step C), collects the results as two separate files for each time series. The reason to save the results for the same meterID into two separate files is to make the data generation model flexible enough to generate synthetic time series with different variances. In this case, the periodic indices could be from a separate time series within the same cluster.

```
1  generate(inputPath, outputPath, frequency, nTimeSeries, nDays,
       baseLoad){
2    PI = readPI(inputPath+"/PI")
3    AR = readAR(inputPath+"/AR")
4
5   results = thetaJoin(PI, AR).get(nTimeSeries)
6       .map((meterId, (coefficients, lagged, PI)) => {
7       newValues = new ARIMAModel(3, 0, 0, coefficients, true)
8         .forecast(lagged, frequency * nDays)
9       .map(x => {
10        reading = x * PI(hour) + baseLoad + Random.nextGaussian()
11        reading })
12       (meterId, newValues)})
13    results.save(outputPath)
14  }
```

Listing 1.2. The code snippet of data generation

The implementation of data generation is shown in Listing 1.2, which takes the resulting models as the input (indicated by inputPath) as well as other parameters including the outputPath, the frequency, the number of time series to generate, the number of days and base load. The program first reads the period indices (PI) and Autogressive models (AR) from the input files into the memory (line 2-3). Then, it does the theta join and returns the desired number of rows (equal to the number of generated time series) (line 5). Third, it does the forecasting using the AR model (line 7-8) and the resulting predicted value is reseasonlized. In addition, the base load and the white noise is also added in the predicted value to simulate reality (line 10). Last, the generated data is written to HDFS (line 13). The synthetic data has the format of *<meter identifier, timestamp, reading>* and an example of the rows is *<100, 201706041900,0.389>*, representing that a meter (with meterID = 100) has used 0.3 kWh electricity in the previous half an hour.

Furthermore, Figure 6 depicts an example of the data generation module. Step A shows the two input files for each meter. Step B shows the shuffling process (from the same cluster) where meterIDs are not equal. The reason to perform this shuffling is to generate numerous time series with several variations using a small seed. Step C shows the forecasted meter readings. Finally (Step D), collects the results as the generated time series.

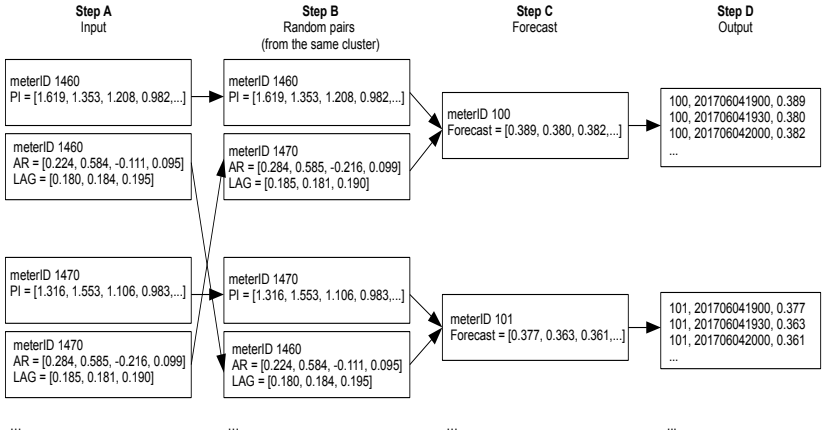

Fig. 6. Data generation module

In general, there are two ways of representing energy consumption. First, a smart energy meter measures a cumulative consumption, i.e., the consumption always increases. Second, a smart meter measures consumption in a given (fixed) interval. i.e, an aggregated value in a time window, e.g., 30 minutes. The generator proposed in this paper is based on the second approach.

4 Evaluation

In this section, we evaluate the data generator in terms of effectiveness and scalability. The effectiveness will be evaluated by comparing the patterns between the real-world and synthetic data. The scalability will be evaluated by measuring the execution performance. The Irish electricity consumption will be used as the seed for training the models.

The experiments are conducted on a 4-node cluster: all the nodes act as slave, and one of them also acts as master. All the machines have the same settings: Intel(R) Xeon(R) CPU E5-2650 (3.40GHz, 4 Cores, hyper-threading is enabled, two hyper-threads per core), 8 GB RAM, and a Seagate Hard driver (1TB, 6 GB/s, 32 MB Cache and 7200 RPM), running 64bit-Ubuntu 12.04 LTS with Linux 3.19.0 kernel.

4.1 Effectiveness

We now evaluate the effectiveness of the proposed smart meter data generator. As mentioned in Section 2.3, the data generator first, uses clustered data as the seed to generate the models, then it generates time series. We use the correlation distance metric for the clustering in the pre-processing of the seed. Before validating the generated time series, we would like to further explain by demonstrating a real example.

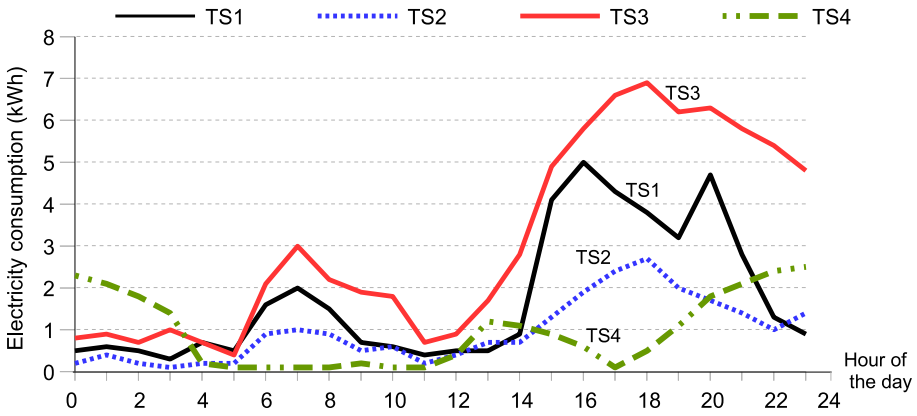

Fig. 7. Daily activity load profile time series

Figure 7, demonstrates four daily load profiles from different households. TS_1 represents a medium energy use household; TS_2 represents a low energy use household, whereas, TS_3 represents a high energy use household. Visually, we could observe that TS_1, TS_2 and TS_3 have a similar pattern, e.g., with morning and evening peaks almost at the same range of the time, although they are within different consumption categories. In contrast, TS_4 is showing a quite different pattern, without morning peak. Hence, according to the consumption patterns, TS_1, TS_2 and TS_3 should be assigned to the same group regardless of their consumption amount, while TS_4 should belong to a different group.

In order to assign the time series to the desired cluster based on the similarity, we compute the distance function. Euclidean function is commonly used as a distance function when performing the clustering. In Section 2.3, we have mentioned that Euclidean function may not give accurate results and we have recommended to use correlation based distance function instead.

Table 1. Comparison of the two distance metrics

	(TS_1, TS_2)	(TS_1, TS_3)	(TS_1, TS_4)	(TS_2, TS_3)	(TS_2, TS_4)	(TS_3, TS_4)
euclDist	6.13	9.12	9.64	11.5	4.73	12.4
corrDist	0.12	0.13	1.06	0.12	0.76	1.10

Table 1, shows the comparison between the two distance functions. If we observe the distances, the correlation distances between (TS_1, TS_2) and (TS_1, TS_3) are smaller than the distance between (TS_1, TS_4). The reason that TS_1, TS_2 and TS_3 have smaller correlation distances is due to the fact that they have similar patterns, whereas, TS_4 has a larger distance for the reason that it has a different pattern with respect to TS_1, TS_2 and TS_3 (note that the distance of zero means perfectly correlated). In contrast, the Euclidean distance between (TS_2, TS_4) is the smallest, which may result in wrongly

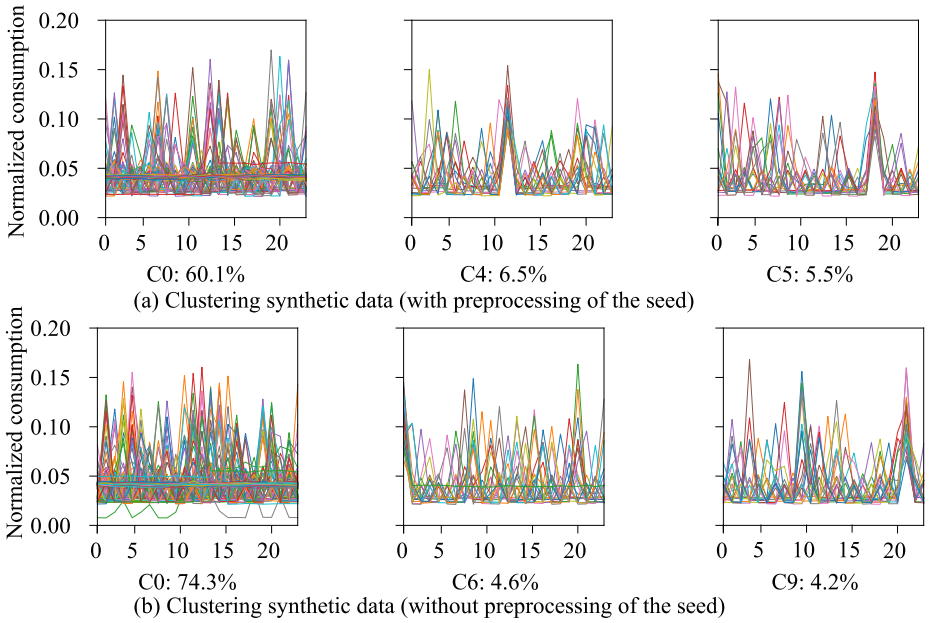

Fig. 8. Comparison of the pattern preservation with and without reprocessing of the seed

assigning TS_4 to the same group as TS_2. Thus, it is more preferable to choose the correlation distance.

We now demonstrate the importance of preprocessing the seed in order to preserve the information of customer segmentation. We compare the clustering information of the resulting synthetic data sets when we use the seed with and without being preprocessed. We cluster the daily patterns into 20 clusters for the two data sets using the adaptive clustering method [16] and compare the top three clusters shown in Figure 8 (a) and (b). According to the top three clusters, we could observe that the patterns are more visible in Figure 8 (a) (where the seed is preprocessed) as compared to Figure 8 (b) (where seed is not preprocessed). Based on these observation, we can conclude that the data generator trained with preprocessed seed can achieve better pattern preservation.

Further, we evaluate the effectiveness by comparing the patterns of the real-world and synthetic data. Figure 9 (a) and (b), show the daily and weekly patterns generated from a typical household, respectively. We compare the patterns of the actual and synthetic data. The synthetic data is generated by the data generators trained by clustered seed using corrDist and euclDist. The actual pattern in Figure 9 (a) shows that there is a morning peak (6-9) and a evening peak (16-21) in the pattern. The pattern of synthetic (corrDist) indicates a good matching to the actual pattern, with very slight drift. In contrast, the matching of synthetic (euclDist) does not show a perfect fit, for example, having a peak at 1-2 o'clock but there is no peak for the actual pattern. Figure 9 (b) shows the weekly patterns, where synthetic (corrDist) also shows better than the synthetic (euclDist) to fit the actual data pattern.

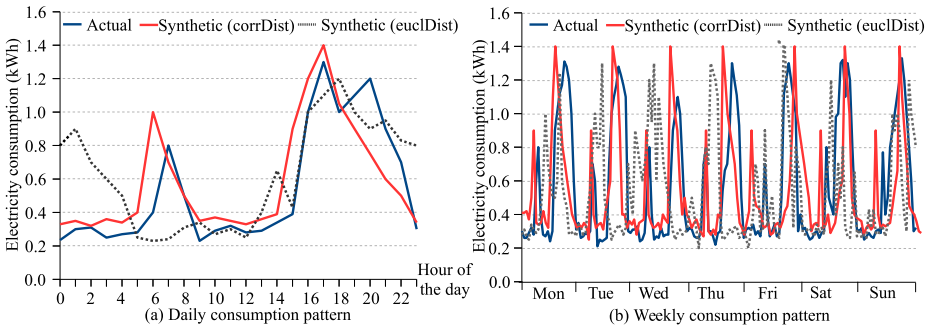

Fig. 9. Comparison of consumption patterns

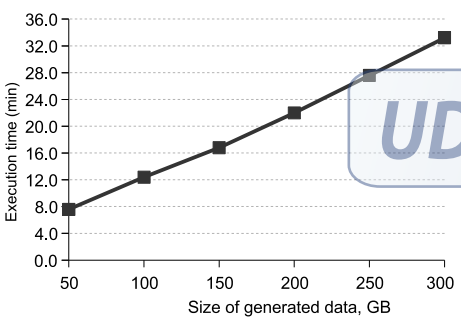

Fig. 10. Scale-up

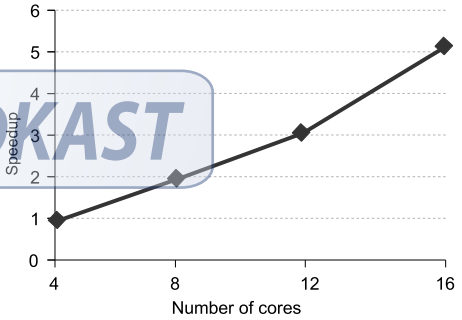

Fig. 11. Speedup of generating 100GB data

4.2 Scalability

In this section, we evaluate the scalability of the proposed data generator. Note that this study will not measure the time of the preprocessing and the training process because they are done only once and the results can be reused many times for data generation. Figure 10, shows the execution time of generating the data scaled from 50 to 300 GB using all nodes (a total of 16 cores). The results show that the execution time increases almost linearly with the size of the data generated.

Figure 11, shows the speedup of generating a fixed size of data set, $100GB$, but varying the number of cores used in Spark. The speedup is calculated as follow: $speedup = t_4/t_n$, where t_4 is the execution time with 4 parallel cores, and t_n is the execution time with n parallel cores (n with the values of $4, 8, 12$ and 16). According to the results, the data generator can achieve a good speedup, when the number of cores increased to 16.

To summarize, the proposed data generator has the ability to generate realistic time series data with a good performance and the generated data has comparable characteristics with the actual data, in terms of patterns and groups/clusters.

5 Related Work

Synthetic data generation has been studied extensively across several disciplines. *DBGEN* is a well-known data generation tool that can generates up to 10 TB of data for the TPC-H/R database schema [17]. Similarly, synthetic weather data generation has also been extensively studied by [18], [19], [20], [21] and [22]. The weather generators typically use stochastic models to simulate synthetic weather data. Furthermore, a vehicle crash data generator uses actual vehicle crash data as seed to produce new realistic data using Fourier transformation [23]. The generated data contains different acceleration peaks to test and verify crash management components in a car without running actual crash tests. Time series forecasting has also attracted much research attention in recent years. A hybrid time series forecasting model based on autoregressive integrated moving average (ARIMA) and neural networks is proposed by [24]. Likewise, a periodic autoregressive moving average model (PARMA) for time series forecasting is also suggested by [25]. PARMA model can explicitly describe seasonal/periodic fluctuations in terms of mean, standard deviation and autocorrelation. Based on that, PARMA derives more realistic time series forecasting models and simulations. In addition, a template-based time series generation tool (loom) that utilizes ARIMA as the underlying forecasting model is presented by [26]. Additionally, a survey is conducted on the forecasting models by [27]. It has reported that ARIMA and neural networks are heavily used in time series forecasting. Based on all these works, it can be concluded that models such as stochastic, ARIMA, PARMA, neural networks play a crucial role in time series forecasting.

In resemblance with these works, the foundation of the proposed data generator is based on autoregressive centered moving average (ARCMA) model.

Smart metering, as an emerging technology has gained widespread attention recently. A lot of work has been reported in the area of smart meter data analytics, however, to the best of our knowledge, the smart meter synthetic data generation still needs to be extensively studied. Some literature has been found with respect to smart meter synthetic data generation by [28] , [2] and [4]. The work by [28], uses Markov chain model, while [2] and [4] use periodic auto-regression (PAR) to generate synthetic time series in order to benchmark Internet of Things (IoT) and smart meter analytics systems.

In contrast to all these works, the focus of the current work is to generate time series based on energy consumption patterns, in a distributed data processing environment.

6 Conclusions and Future Work

Smart meter data management and analytics systems require a large amount of data for benchmarking and testing purposes. In this paper, we have presented a scalable smart meter data generator using the Spark framework. We have used the supervised machine learning method to create the models for generating synthetic data. In addition, we have introduced an optimization method that preserves user-groups/clusters information, i.e., using clustered seeds. We have comprehensively evaluated the data generator by comparing its effectiveness and scalability. The results have demonstrated that the data generator can generate scalable smart meter data that can simulates well to the reality.

For the future work, we could consider to add more features to the data generation models, for example, seasonality that is winter, spring, summer and autumn patterns. In addition, the current generator could be extended or modified to generate other types of meter data, such as water, gas and heating.

Acknowledgement. This research is supported by UCN-FOU funding (Project-6/2016-17) and the CITIES project by Danish Innovation Fund (1035-00027B).

References

1. Smart Meter From Wikipedia, en.wikipedia.org/wiki/Smart_meter
2. Liu, X., Golab, L., Golab, W., Ilyas, I. F.: Benchmarking Smart Meter Data Analytics. In: Proc. of the 18th International Conference on Extending Database Technology, pp. 385–396 (2015)
3. Liu, X., Golab, L., Golab, W., Ilyas, I. F., Jin, S.: Smart Meter Data Analytics: Systems, Algorithms, and Benchmarking. In: ACM Transactions on Database Systems (TODS), 42(1), Article 2. ACM Press, NY (2017)
4. Liu, X., Golab, L., Ilyas, I. F.: SMAS: A Smart Meter Data Analysis System (demo). In: Proc. of the 31st International Conference on Data Engineering, pp. 147–1479 (2015)
5. ISSDA, www.ucd.ie/issda/data/commissionforenergyregulationcer
6. Iftikhar, N., Liu, X., Nordbjerg, F. E., Danalachi, S.: A Prediction-based Smart Meter Data Generator. In: 19th International Conference on Network-Based Information Systems, pp. 173–180, IEEE (2016)
7. Time Series Components, www.otexts.org/fpp/6/1
8. Zhang, G. P., Qi, M.: Neural Network Forecasting for Seasonal and Trend Time Series. European Journal of Operational Research, 160(2), 501–514 (2005)
9. Weiers, R.: Introduction to Business Statistics. Cengage Learning (2010)
10. Lawrence, K. D., Klimberg, R. K., Lawrence, S. M.: Fundamentals of Forecasting using Excel. Industrial Press Inc. (2009)
11. Okcan, A., Riedewald, M.: Processing theta-joins using MapReduce. In: Proc. of SIGMOD, pp. 949-960 (2011)
12. Wu, J.: Advances in K-means Clustering: A Data Mining Thinking. Springer Science & Business Media (2012)
13. Parsian, M.: Data Algorithms: Recipes for Scaling Up with Hadoop and Spark. O'Reilly Media, Inc. (2015)
14. Liao, T. W.: Clustering of Time Series DataA Survey. Pattern Recognition, 38(11), 1857–1874 (2005)
15. Black, K.: Business Statistics: For Contemporary Decision Making. John Wiley & Sons (2011)
16. Peng, B., Wan, C., Dong, S., Lin, J., Song, Y., Zhang, Y., Xiong, J.: A Two-stage Pattern Recognition Method for Electric Customer Classification in Smart Grid. In: Smart Grid Communications (SmartGridComm), pp. 758–763 (2016)
17. Poess, M., Floyd, C.: New TPC Benchmarks for Decision Support and Web Commerce. ACM Sigmod Record, 29(4),64–71 (2000)
18. Breinl, K., Turkington, T., Stowasser, M.: Simulating Daily Precipitation and Temperature: A Weather Generation Framework for Assessing Hydrometeorological Hazards. Meteorological Applications, 22(3), 334–347 (2014)

19. Li, Z., Brissette, F., Chen, J.: Finding the Most Appropriate Precipitation Probability Distribution for Stochastic Weather Generation and Hydrological Modeling in Nordic Watersheds. Hydrological Processes, 27(25), 3718–3729 (2013)
20. Breinl, K., Turkington, T., Stowasser, M.: A Weather Generator for Hydro-meteorological Hazard Applications EGU General Assembly Conference. In EGU General Assembly Conference Abstracts, vol.16, p. 10522 (2014)
21. van Paassen, A. H., Luo, Q. X.: Weather Data Generator to Study Climate Change on Buildings. Building Services Engineering Research and Technology, 23(4):251–258 (2002)
22. Shamshad, A., Bawadi, M. A., Hussin, W. W., Majid, T. A., Sanusi, S. A. M.: First and Second Order Markov Chain Models for Synthetic Generation of Wind Speed Time Series. Energy, 30(5), 693–708 (2005)
23. Cuddihy, M. A., Drummond Jr, J. B., Bourquin, D. J.: Ford Motor Company, Vehicle Crash Data Generator. U.S. Patent No. 5,608,629 (1997)
24. Zhang, G. P.: Time Series Forecasting using a Hybrid ARIMA and Neural Network Model. Neurocomputing, 50, 159–175 (2003)
25. Anderson, P. L., Meerschaert, M. M., and Zhang, K.: Forecasting with Prediction Intervals for Periodic Autoregressive Moving Average Models. Journal of Time Series Analysis, 34(2), 187–193 (2013)
26. Kegel, L., Hahmann, M., Lehner, W.: Template-based Time Series Generation with Loom. In: EDBT/ICDT Workshops, 1558 (2016)
27. De Gooijer, J. G., Hyndman, R. J.: 25 Years of Time Series Forecasting. International Journal of Forecasting, 22(3), 443–473 (2006)
28. Arlitt, M., Marwah, M., Bellala, G., Shah, A., Healey, J., Vandiver, B.: IoTA Bench: An Internet of Things Analytics Benchmark. In: 6th ACM/SPEC International Conference on Performance Engineering, pp. 133–144, ACM Press, NY (2015)