SEAI RD&D 2019

SMARTeBuses

Big Data Analytics With Fast and Scalable Access to Historical Data

Milestone number: WP2-M1

This Project has received funding from the SEAI RD&D Programme.
Grand Award Number: 19/RDD/519

Project Acronym: SMARTeBuses
Project Full Title: SMART electric Buses
Grant Number: 19/RDD/519
Project URL: https://smartebuses.github.io/web/

Deliverable type:	Milestone / Report
Dissemination level:	Private
Delivery Date:	October 31st, 2020
Number of pages:	17
Keywords:	SMARTeBuses, Artificial Intelligence, Big Data
Authors:	Dr. Alejandro Arbelaez - University College Cork Dr. Ignacio Castiñeiras - Cork Institute of Technology
Peer review:	Padraigh Jarvis - University College Cork

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

SEAI	Sustainable Energy Authority of Ireland
SMARTeBuses	SMART electric Buses
EU	European Union
AI	Artificial Intelligence
HDFS	Hadoop Distributed File System
RDD	Resilient Distributed Datasets
API	Application Programming Interface

1 Introduction

This document corresponds to the milestone report WP2-M1 *"Big Data Analytics With Fast and Scalable Access to Historical Data"* of the SMART electric Buses (SMARTeBuses) Project, funded by the Sustainable Energy Authority of Ireland (SEAI) RD&D programme. This project is classified as Non-economic public Good Research under the European Union (EU) State Aid regulations and will exploit, combine and improve cutting-edge Artificial Intelligence (AI) technologies to develop and implement optimization models for the operation of electric buses in Ireland with operational constraints.

In this milestone, we focus our attention in the use of Spark to analyse historical data from two sources: (i) the transportation sector (i.e., 208 bus route in Cork city) and (ii) the energy production in the Ireland (including wind energy). We recall that this information will play an important role for future work packages, e.g., to identify suitable locations for fast charging stations, the scheduling of charging/discharging events for the electric fleet, and forecasting the production of wind energy in Ireland.

We recall that while historic information on the location of the buses of the 208 route in Cork is not available online, real-time information can be obtained through the TFI Real Time Ireland App available at https://www.transportforireland.ie/available-apps. Therefore, in this report, we use Big Data technologies to analyse data from mid June 2016 to late September 2017. The dataset was created by querying the Application Programming Interface (API) every three minutes from 6:30AM to midnight and gathering all entries of a day into a file.[1] Likewise we use a public data available at [1] for our energy production case study.

As indicated in [2], the SMARTeBuses project uses a set of Big Data storage and analysis tools, including Hadoop Distributed File System (HDFS), Hadoop MapReduce and Spark. In this milestone, we focus in two Spark libraries for our big data analysis, i.e., Spark Core and Spark SQL. Whereas Spark Core uses Resilient Distributed Datasets (RDD) as its main data abstraction, Spark SQL uses DataFrames. Taking this into account, Chapter 2 presents our Big Data analysis for the public transportation dataset using Spark Core and Chapter 3 presents our Big Data analysis for the energy production dataset using Spark SQL.

[1]We would like thank Michael O'Keefe and the Insight Center for Data Analytics for sharing this dataset.

2 Big Data & Public Transportation

The transportation sector generates large quantities of data with real-time updates of the service. [3, 4] outlines the importance of Big Data to store and process the increasingly available data, such as: time-stamped and geo-tagged location of the buses, i.e., latitude and longitude coordinates of the buses. In the SMARTeBuses project, we focus our attention in the Irish transportation system and for this milestone we use the 208 bus route in Cork city for our case study.

In this chapter, we provide a set of algorithms implemented in Spark Core (using the PySpark library with RDD) to compute general statistics about the 208 bus route in Cork city from mid June 2016 to late September 2017. The RDD main data abstraction of Spark Core handles distributed objects with multiple features, including but not limited to: lazy evaluation, fault-tolerance, and in-memory RAM computation.

In our 208 route dataset, each observation or datapoint has the following structure:

- Station id
- Direction
- Day of the week
- Timestamp of the query
- Scheduled arrival time
- Expected arrival time

Furthermore, in this milestone we use the following stations for our this case study:

- Station 240101 (UCC WGB - Lotabeg)
- Station 240561 (UCC WGB - Curraheen)
- Station 241111 (CIT Technology Park - Lotabeg)
- Station 240491 (Patrick Street - Curraheen)

The dataset provides regular updates of the arrival time of the buses to each (selected) station and we focus in the following time intervals: Morning from 7:00 AM to 10:00 AM and Evening from 4:00PM to 7:00PM.

2.1 Number of observations in the dataset

In this section, we start our analysis with a basic query that list the number of observations per query. We would like to recall that even thought the goal of the bus operators if to provide full digital map of the service, only a fraction of the buses currently allow a real-time tracking of the system.

Algorithm 2.1 describes the simplest analysis of the dataset, calculating the number of observations in the system. In Line 2 the SparkContext (i.e., sc) loads the dataset from the HDFS and scans each observation; line 3 counts the number of observations; and line 4 displays the final result.

Algorithm 2.1: Observations per station

```
def observations_per_station(sc, my_dataset_dir):
    inputRDD = sc.textFile(my_dataset_dir) #Loading the dataset
    resVAL = inputRDD.count()              #Counting
    print(resVAL)                          #Displaying the results
```

2.2 Buses arriving ahead or behind schedule

One of the main goals of the SMARTeBuses is to identify suitable locations for fast charging stations. In this context, WP-3 will develop optimization algorithms to minimize the number of charging units while satisfying the demand and maximizing the use of renewable energies. In this context, we aim at synchronising the charging and discharging times with the current working schedule of the buses.

Algorithm 2.2 describes the general steps to process observations from the raw data and facilitate subsequent operations. In particular, we remove observations with inconsistent or missing data.

Algorithm 2.2: Processing Raw Data

```
def process_line(line):
    res=()
    line=line.replace(”\n”, ””)
    params = line.split(”;”)     #Input structure pre-processing
    if (len(params)==8):
        res=(int(params[0]), str(params[1]), str(params[2]),
                str(params[3]), str(params[4]),str(params[5]),
                str(params[6]), str(params[7]) )
    return res
```

Algorithm 2.3: Total Buses Behind and Ahead Schedule

```
1 def ahead_or_behind_schedule(sc, my_dataset_dir, st_number):
2     inputRDD = sc.textFile(my_dataset_dir)              #Loading the dataset
3     linesRDD = inputRDD.map(process_line)               #Processing lines
4     stationRDD = linesRDD.filter(
         lambda line: line[0] == st_number)               #Relevant stations
5     infoRDD = stationRDD.map(
         lambda line: (line[6], line[7]))                 #Relevant data
6     on_timeRDD = infoRDD.map(
         lambda line: (1 if line[0] >= line[1] else 0,
                       1 if line[0] <  line[1] else 0) )  #Buses ahead and behind
7     resVAL = on_timeRDD.reduce(lambda x, y:
         (x[0] + y[0], x[1] + y[1]) )                     #Post-processing
8     print(resVAL)                                       #Displaying the solution
```

Algorithm 2.3 outlines the Spark implementation to identify the number the number of buses ahead or behind original time in a given station (i.e., st_number). In this algorithm line 2 loads the dataset from the HDFS. Line 3 transforms the observations from the raw dataset into a new RDD by applying the function *process line* (see Algorithm 2.2). Line 4 filters out irrelevant stations. Lines 5-6 select relevant data for this particular query, i.e., scheduled and expected arrival time, to calculate the number of buses ahead or behind the schedule. Finally, line 7 aggregates the partial results to compute the final numbers.

Table 2.1 shows the number of buses ahead and behind the original schedule. As it can be observed Stop 240491 reports the largest number of buses behind the schedule and 20561 reports the largest number of buses ahead the original schedule.

Stop	Behind	Ahead
240491	10122	6293
240171	5465	7177
240101	5337	10415
240561	4987	11427
240001	3840	9276
241111	5698	7033

Table 2.1: Number of buses ahead and behind the schedule

2.3 List of buses scheduled for each day of the week

In addition to identifying potential charging/discharging events for the buses we also need to identify a mapping between the buses and the stops for each day of the week. Algorithm 2.4 outlines the process to calculate the number of buses scheduled to use a given station for each day of the week. Similarly to the previous algorithm, Lines 2 to 5 loads the dataset, identify the relevant stations and the relevant data. Line 6 identifies the buses using the station. Lines 7 and 8 aggregates the data by weekly and days of the week.

Algorithm 2.4: Scheduled buses per day of the week

```
1  def buses_per_day(sc, my_dataset_dir, station_number):
2      inputRDD = sc.textFile(my_dataset_dir)        #Loading the dataset
3      linesRDD = inputRDD.map(process_line)         #Pre-processing
4      stationRDD = linesRDD.filter(
         lambda line: line[0] == station_number)     #Selecting the st.
5      infoRDD = stationRDD.map(
         lambda line : (line[3] + ” ” + line[6]) )   #Selecting relevant data
6      repeatedRDD = infoRDD.distinct()              #Different buses
7      keyRDD = repeatedRDD.map(lambda line:
         (line.split(” ”)[0], line.split(” ”)[1]))
8      groupRDD = keyRDD.groupByKey()                #Aggregating daily data
9      solutionRDD = groupRDD.mapValues(my_sort)     #Sorting the result
10     resVAL = solutionRDD.collect()                #Collecting the results
11     for item in resVAL:
12         print(item)                               #Displaying the results
```

In the following we describe the scheduled timetable of the buses for a subset of the stops:

Stop: 240491 - St. Patrick Street - Curraheen

Weekdays	→	7:10, 7:14, 07:20, 7:31, 7:39, 7:49, 7:59, 8:09, 8:29, 8:39, 8:49, 8:59, 9:09, 9:16, 9:19, 9:26, 9:36, 9:46, 9:56, 10:06
Saturdays:	→	7:10, 7:25, 7:45, 8:05, 8:25, 8:45, 9:05, 9:25, 9:51, 10:11
Sundays:	→	9:20, 9:54, 10:14

Stop: 240561 - UCC WGB - Curraheen

Weekdays	→	7:18, 7:26, 7:32, 7:43, 7:51, 8:01, 8:11, 8:21, 8:41, 8:51, 9:01, 9:11, 9:21, 9:25, 9:31, 9:35, 9:45, 9:55, 10:05
Saturdays:	→	7:16, 7:31, 7:51, 8:11, 8:31, 8:51, 9:11, 9:31, 9:59, 10:19
Sundays:	→	9:27, 10:02

Stop: 240001 - CIT Technology Park - Curraheen

Weekdays	→	7:08, 7:28, 7:42, 7:48, 7:59, 8:07, 8:17, 8:27, 8:37, 8:54, 9:04, 9:14, 9:24, 9:34, 9:37, 9:44, 9:47, 9:57, 10:07
Saturdays:	→	7:05, 7:25, 7:40, 8:00, 8:20, 8:40, 9:00, 9:20, 9:40, 10:12
Sundays:	→	9:37, 10:11

Stop: 241111 - CIT Technology Park - Lotabeg

Weekdays	→	16:01, 16:11, 16:21, 16:31, 16:45, 16:51, 17:01, 17:11, 17:21, 17:31, 17:45, 17:51, 18:01, 18:11, 18:21, 18:31, 18:41, 18:51, 19:11
Saturdays	→	16:01, 16:11, 16:21, 16:31, 16:41, 16:51, 17:01, 17:11, 17:21, 17:31, 17:41, 17:51, 18:01, 18:11, 18:31, 18:51, 19:11
Sundays	→	16:11, 16:31, 16:51, 17:11, 17:31, 17:51, 18:11, 18:31, 18:51, 19:11

Stop: 240101 - UCC WGB - Lotabeg

Weekdays	→	16:00, 16:06, 16:16, 16:26, 16:36, 16:46, 16:58, 17:04, 17:14, 17:24, 17:34, 17:44, 17:58, 18:04, 18:14, 18:24, 18:34, 18:44, 18:54, 19:04
Saturdays	→	16:03, 16:13, 16:23, 16:33, 16:43, 16:53, 17:03, 17:13, 17:23, 17:33, 17:43, 17:53, 18:03, 18:12, 18:22, 18:42, 19:02
Sundays	→	16:04, 16:24, 16:44, 17:04, 17:24, 17:44, 18:04, 18:24, 18:44, 19:04

Stop: 240171 - Patrick Street - Lotabeg

Weekdays	→	16:03, 16:13, 16:23, 16:33, 16:43, 16:53, 17:03, 17:10, 17:20, 17:30, 17:40, 17:50, 18:00, 18:10, 18:20, 18:30, 19:00
Saturdays	→	16:06, 16:16, 16:26, 16:36, 16:46, 16:56, 17:06, 17:16, 17:26, 17:36, 17:46, 17:56, 18:06, 18:22, 18:52, 19:12
Sundays	→	16:15, 16:35, 16:55, 17:15, 17:35, 17:55, 18:15, 18:35, 18:55, 19:15

2.4 Average waiting times

Algorithm 2.5 describes the process to calculate the average daily and monthly waiting times. Lines 2 and 3 load and pre-process the observations in the raw dataset. Lines 4-5 filter out irrelevant data. Lines 5 to 7 aggregate the average daily for the indicated months. Lines 10-12 collect and display the final output.

Algorithm 2.5: Average Waiting Times week

```
def average_waiting_time(sc, my_dataset_dir, st_number, month_list):
    inputRDD = sc.textFile(my_dataset_dir)          #Loading the dataset
    linesRDD = inputRDD.map(process_line)           #Pre-processing
    stationRDD = linesRDD.filter(
        lambda line: (line[0] == st_number) and
              (line[4][3:5] in month_list) )        #Selecting the st.
    infoRDD = stationRDD.map( lambda line : (line[3] + " " +
            line[4][3:5], (line[5], line[7])) )     #Selecting relevant data
    waiting_timeRDD = infoRDD.mapValues( lambda val:
     time_to_int(val[1])-time_to_int(val[0]))       #Waiting time per observation
    aggregatedRDD = waiting_timeRDD.combineByKey(
              lambda x:(x,1),
              lambda x,y:(x[0]+y,x[1]+1),
              lambda x,y:(x[0]+y[0],x[1]+y[1]))     #Daily and monthly waiting time
    averageRDD = aggregatedRDD.mapValues(
              lambda x: float(x[0])/float(x[1]))    #Avg. daily and monthly data
    solutionRDD = averageRDD.sortBy(lambda item: item[1])
    resVAL = solutionRDD.collect()                  #Aggregating data
    for item in resVAL:
        print(item)                                 #Displaying the results
```

Figure 2.1 summaries the average daily waiting times for each month. Interestingly, the average is consistently higher during the weekends than during the weekdays. September and October are the months with higher daily waiting times with an average waiting time of more than 20 minutes on Sundays. July reports the daily with the lowest average waiting time with about 2.6 minutes on Wednesdays.

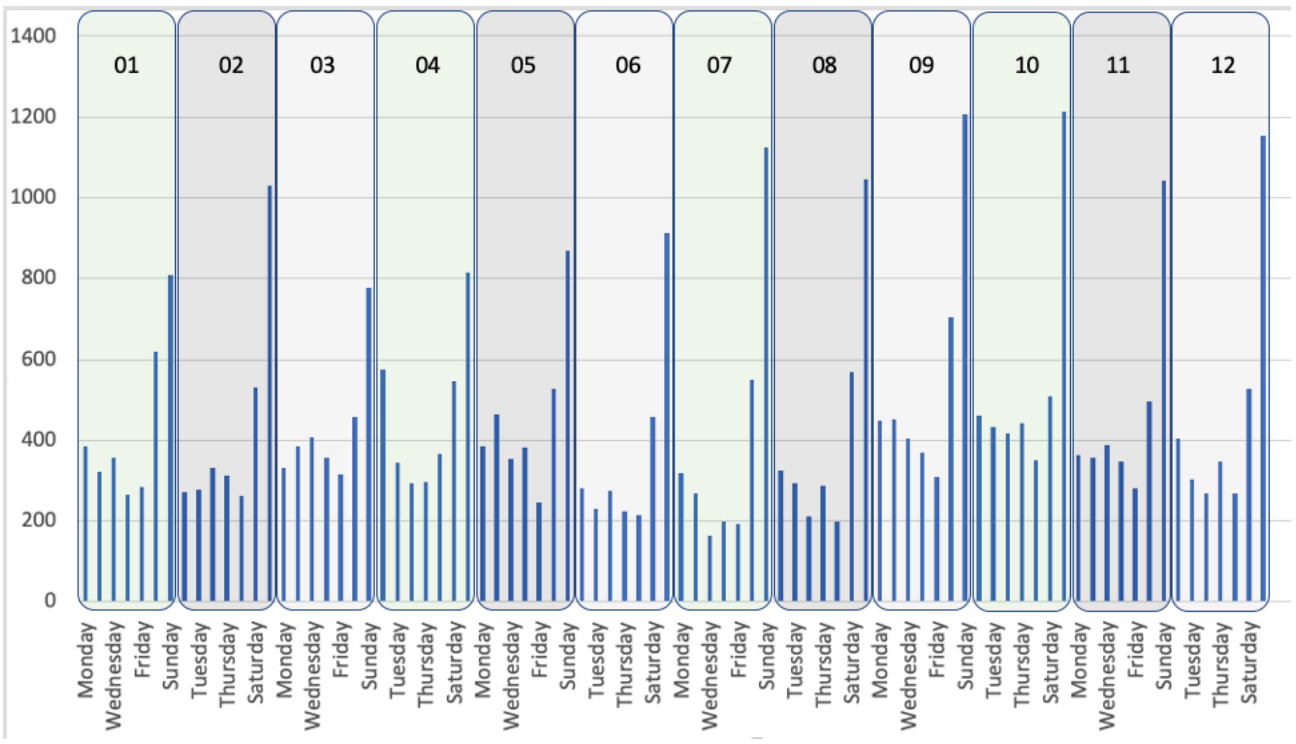

Figure 2.1: Daily waiting time for each month of the year - Stop: 240491

3 Big Data & Wind Power

As of 2017 renewable energies contributed with about 9% of the energy needs in Ireland with a national target of 40% by 2020 [5]. In this milestone, we provide an initial investigation of the current availability of wind energy in Ireland. We recall that wind energy is intermittent and only available when certain meteorological conditions are met. We provide a set of algorithms in Spark SQL (using the PySpark library with DataFrames) to compute general statistics about the amount of wind energy vs. total energy needs in Ireland. In Spark SQL, a DataFrame abstraction handles distributed tables containing containing a number of rows, each one with a number of columns.

In this milestone we analyse the energy demand of this year (until the end of August). The dataset has been collected in 15-minute intervals. [6] reports the energy needs in Ireland (e.g., energy demand) with three columns, i.e., date, actual energy consumption in kWh, and the region (i.e., Republic of Ireland, Northern Ireland, and the entire island). Likewise [6] reports the amount of wind power with four columns, i.e., date, expected wind power, actual wind power, and region. The expected amount of wind energy helps the national grid to decide to switch off the turbines when the supply of energy exceeds the demand or to prepare contingency plans to supply the demand in the absent of enough power from the wind farms.

3.1 Monthly Average

In this section, we analyse the monthly average energy needs of Ireland vs. the available wind energy. Algorithm 3.1 provides a description of the pre-processing method to read and load the dataset from the HDFS into Spark SQL as a DataFrame. Lines 2 and 3 define the structure of the input data, Lines 4 -9 load the dataset into the demand/wind dataframes and remove missing and duplicated data. Lines 10-13 re-defines the date attributes in order to filter out observations outside the relevant time window.

Algorithm 3.1: Loading Wind Data

```
1def ReadDataFrames(sc, sd_dir, wind_dir):
2  my_schema_demand = StructType( [
     StructField(”date”,  TimestampType(),True),
     StructField(”Actual”,IntegerType(),  True),
     StructField(”Region”,StringType(),   True)])     #System Generation Structure
3  my_schema_wind = StructType([
     StructField(”date”,     TimestampType(),True),
     StructField(”forecast”,IntegerType(),   True),
     StructField(”Actual”,   IntegerType(),   True),
     StructField(”Region”,   StringType(),    True)]) #Wind Generation Structure
4  demandDF = spark.read.format(”csv”).option(”delimiter”, ”,”).
     option(”header”,”true”).
     option(”timestampFormat”,”dd MMMM YYYY H:mm”).
     schema(my_schema_demand).load(sd_dir)            #Loading Data
5  windDF = spark.read.format(”csv”).option(”delimiter”, ”,”).
     option(”header”,”true”).
     option(”timestampFormat”,”dd MMMM YYYY HH:mm”).
     schema(my_schema_wind).load(wind_dir)            #Loading Data
6  windDF = windDF.dropna()                            #Missing data
```

```
windDF = windDF.dropDuplicates(["date","Actual"])
demandDF = demandDF.dropna()                              #Missing Data
demandDF = demandDF.dropDuplicates(["date","Actual"])
demandDF = demandDF.withColumn("ndate",
    date_format("date", "MM/dd/YYYY"))                    #Removing Minutes and Hours
demandDF = demandDF.filter(demandDF["date"].
    between('2020-01-01','2020-09-01'))                   #Evaluation timeframe
windDF = windDF.withColumn("ndate",
    date_format("date","MM/dd/YYYY"))                     #Removing Minutes and Hours
windDF = windDF.filter( windDF["date"].
    between('2020-01-01','2020-09-01') )                  #Evaluation timeframe
return [demandDF, windDF]
```

Algorithm 3.2 shows the spark implementation of the monthly average demand vs. wind energy. The algorithm uses the groupby method to split the data into groups based on the month of the year. As it can be seen in Figure 3.1 January, February, and March are the months with higher energy consumption, while May is the month with the lowest energy consumption. As expected, the wind energy production is not following the same patter as the energy consumption due to the variability of the meteorological conditions in the year. Typically, the wind speed is higher during the winter months and decreases towards the summer. Our findings indicate that February (resp. September) is the month with highest (resp. lowest) production of wind energy.

It is important to recall that even though Figure 3.1 displays a significant drop in wind energy production in March during the COVID-19 lockdown, we do not have enough evidence that this negatively impacted the production of wind energy in Ireland. We rather attribute the lower wind energy production in March to the annual seasonal patters. However a deep analysis, outside of the scope of the project, would be required in order to statistically validate the impact of the lockdown in the wind energy production in Ireland.

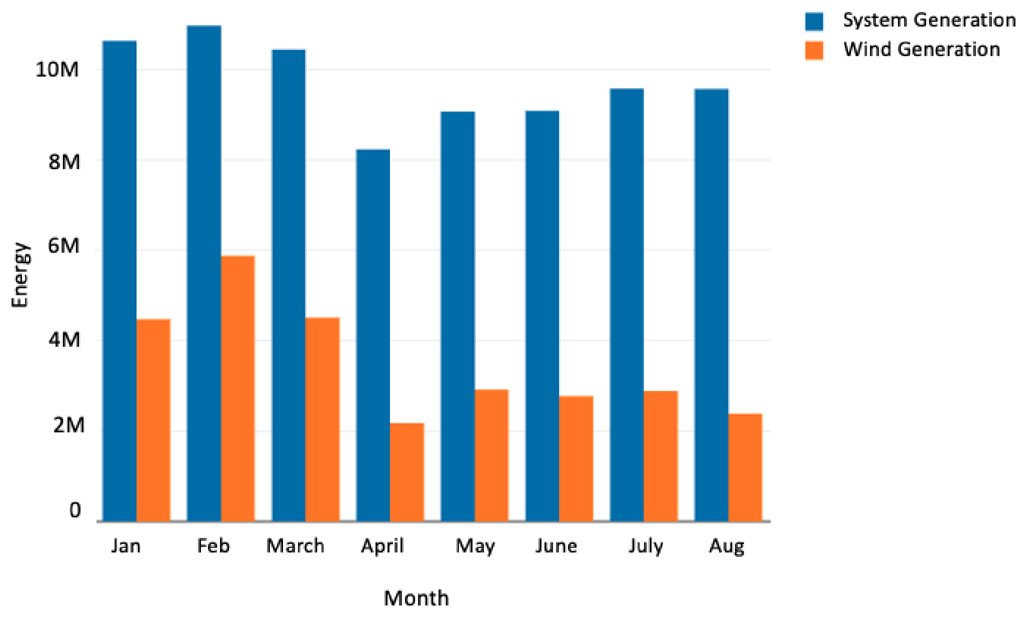

Figure 3.1: Monthly Energy (2020)

Algorithm 3.2: Monthly Average

```
def MonthlyAverage(sc, sd_dir, wind_dir):
  [demandDF, windDF] = ReadDataFrames(sc,sd_dir,wind_dir)      #Loading Data
  demandDF = demandDF.withColumn(”ndate”,
     date_format(”date”, ’MM/YYYY”))                           #Reformat date
  demandDF = demandDF.groupBy(”ndate”).agg(sum(”Actual”).
     alias(”Daily SG”)).sort(’ndate’)
  windDF = windDF.withColumn(”ndate”,
     date_format(”date”, ’MM/YYYY”))                           #Reformat date
  windDF = windDF.groupBy(”ndate”).agg(sum(”Actual”).
     alias(”Daily Wind”)).sort(’ndate’)
  fullDF = windDF.join(demandDF, windDF.ndate==demandDF.ndate) #concat dataframes
  return fullDF
```

3.2 Daily Average

In this section, we describe our Spark implementation of the daily average energy production in Ireland. Algorithm 3.3 calculates the moving average by computing the average over a specific number of days, lines 3-5 calculates the total daily energy production, line 5 defines the moving average timeframe, and lines 6-8 calculate the moving average for both the energy demand and wind energy production.

Algorithm 3.3: Daily Moving Average

```
1 def DailyAverage(sc, sd_dir, wind_dir):
2   [demandDF, windDF] = ReadDataFrames(sc, sd_dir, wind_dir) #Loading data
3   demandDF = demandDF.groupBy(”ndate”).
     agg(sum(”Actual”).alias(”Daily”)).sort(’ndate’).      #Aggregating energy demand
4   windDF = windDF.groupBy(”ndate”).
     agg(sum(”Actual”).alias(”Daily”)).sort(’ndate’).      #Aggregating wind energy
5   w = Window.orderBy(’ndate’).rowsBetween(-6, 0)         #7-day timeframe
6   demandDF = demandDF.withColumn(’System’, avg(’Daily’).over(w))#Averaging demand
7   windDF = windDF.withColumn(’Wind’, avg(’Daily’).over(w)) #Averaging wind energy
8   fullDF = windDF.join(demandDF, windDF.ndate==demandDF.ndate) #Concat dataframes
9   return fullDF
```

Figure 3.2 displays the energy production in Ireland for the indicated timeframe. As expected, the wind energy production is considerably lower than the demand and highly variable. This figure is consistent with [6], however, [6] aggregates the energy supply and demand from different sources (e.g., oil, gas, petrol, etc), while we report the data as generated in [1]. Similarly to [6] we observe a lower energy demand after the COVID-19 lockdown in March. However, as previously indicated for the monthly wind energy production additional a deep analysis would be required to study the impact of the lockdown in the wind energy production.

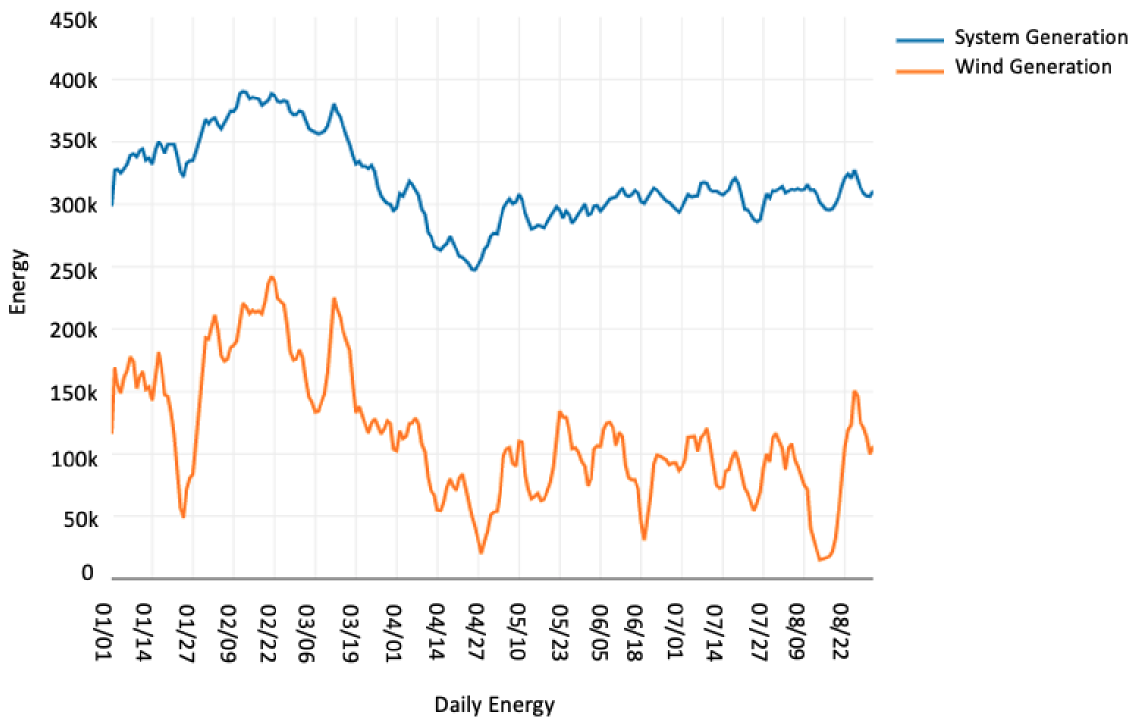

Figure 3.2: Daily Moving Average (2020)

3.3 Hourly Average

In this section, we describe our Spark implementation of the hourly energy production in Ireland. We recall that the one of the goals of the SMARTeBuses project is to maximise the use of wind energy for the transition to eBuses. In this context, we need to identify suitable charging and discharging times of the buses, therefore, we need to investigate times of the day with low energy demand and large amounts of wind energy production.

Algorithm 3.4: Daily Moving Average

```
def HourlyAverage(sc, sd_dir, wind_dir):
  [demandDF, windDF] = ReadDataFrames(sc, sd_dir, wind_dir) #Loading data
  windDF = windDF.withColumn("HFormat",
    date_format("date", "dd/MM/YYYY HH"))
  windDF = windDF.withColumn("HDate", hour(col("date")))     #Hourly format
  windDF = windDF.groupBy(["HFormat", "HDate"]).
    agg(sum("Actual").alias("Hourly")).sort("HFormat")      #Aggregating energy
  windDF = windDF.groupBy("HDate").
    agg(avg("Hourly").alias("Hourly Wind")).sort("HDate")   #Hourly format
  demandDF = demandDF.withColumn("HFormat",                 #Averaging
    date_format("date", "dd/MM/YYYY HH"))
  demandDF = demandDF.withColumn("HDate", hour(col("date")))
  demandDF = demandDF.groupBy(["HFormat", "HDate"]).
    agg(sum("Actual").alias("Hourly")).sort("HFormat")      #Aggregating energy
  demandDF = demandDF.groupBy("HDate").
    agg(avg("Hourly").alias("Hourly SG")).sort("HDate")     #Averaging
  fullDF = windDF.join(demandDF, windDF.HDate==demandDF.HDate) #Concat dataframes
  return fullDF
```

Algorithm 3.4 outlines our algorithm. The algorithm starts by reformatting the date attribute to subsequently split and aggregate hourly data. Figure 3.4 displays the outcome of the algorithm and it can be observed that the energy demand peaks at about 6:00PM and 4:00AM is the hour with the lowest demand. On the other hand, the wind energy production exhibits lower variation with a peak between 2:00PM and 4:00PM. However, we would like to remark that our algorithm takes into account the overall average the time period. We plan to further analyse the hourly consumption at different times of the year.

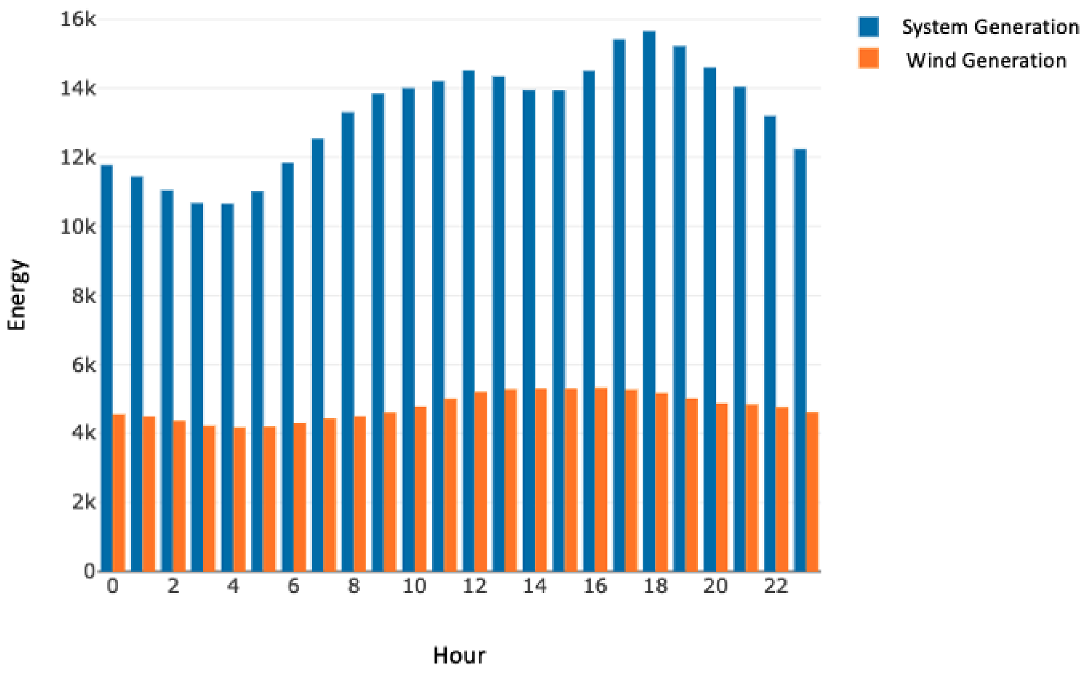

Figure 3.3: Hourly Average

4 Conclusions

In this milestone report we have described an set of algorithms implemented in Spark Core and Spark SQL for a fast and scalable analysis of historical data of the 208 bus route in Cork city and the amount of energy production (including wind energy) in Ireland. We plan to extend this initial analysis in W2-D2 (*General Analysis of Historical Data*) with additional algorithms for the analysis of a larger dataset for the bus network in Dublin. The Dublin dataset is considerably bigger than the current study.

Bibliography

[1] EirGrid Group, “Smart Grid Dashboard,” http://smartgriddashboard.eirgrid.com, Accessed on 2020-10-30.

[2] I. Castiñeiras, “Deliverable WP2-D1, storing data in a hadoop cluster,” Aug 2020, tech. Rep. WP2-D1.

[3] S. Rusitschka and E. Curry, *Big Data in the Energy and Transport Sectors*. Springer, 2016, pp. 225–244.

[4] S. Campos-Cordobes, J. del Ser, I. Laña, I. Olabarrieta, J. Sanchez-Cubillo, J. J. Sanchez-Medina, and A. L. Torre-Bastida, *Big Data in Road Transport and Mobility Research*. Elsevier, 2018, pp. 175–205.

[5] SEAI - Sustainable Energy Authority of Ireland, “Renewable energy in ireland - 2019 report,” https://www.seai.ie/publications/Renewable-Energy-in-Ireland-2019.pdf.

[6] ——, “Tracking effect of covid-19 on energy supply and demand,” May 2020, Tech. Rep.

SMARTeBuses

SMART electric Buses

October 27, 2020