MIE1628 Final Project Report

Group Number: 8

Student Name: Zijian Wei

Student Number: 1002276823

Submission Data: Dec 22nd, 2020

Contents

1. Dataset Selection and Size

My group chose Gbatteries project and the dataset is offered rather than selected. The reason of choosing such project is that we want to attempt an actual problem in industry and apply what we have learned in this course. With this project, we are solving a challenge that can bring real benefits to industrial process, which is exciting. We also think that such dataset offered has enough complexity for us to work with.

The raw data set has 32266 records in total for 48 batteries, and 10 fields. The file size of the raw data in .csv format is 2.8 MB. The size of dataset is small. The details of dataset are discussed later after defining the problem.

2. Problem Definition

2.1 Background Information

Nowadays, rechargeable battery plays an important role in many different fields for its ability of storing electrical energy and recharging. One of the measurements of performance of a rechargeable battery is its durability, which can be represented by the number of time it can be recharged while still being able to release enough electrical energy.

In this Project, the metric of durability is the number of cycles that a battery is charged and then discharged before reaching its end of life (EOL). There are two types of cycling: regular cycle and RPT cycle. RPT cycles are specialized cycles to measure the retentive capacity of battery. EOL happens when the RPT discharge capacity crosses 80% of the nominal value of capacity.

Testing durability of batteries with cycling is a lengthy process and can take months. Therefore, being able to forecast the RPT discharge capacity accurately can greatly shorten the durability testing process.

2.2 Scoping

The retentive capacity of battery has a general decreasing trend as cycle number increases. Although RPT cycle is specialized cycle to measure the retentive capacity accurately, it can sometimes be volatile. In order to determine the EOL correctly, the trend of RPT discharge capacity is focused regardless of volatility. Notice that volatility is complex and can differ significantly depending on the set of charging parameters, which are not given in the data and are different for each unique battery id. The dataset is of small size and would not satisfy the need of investigating volatility. In addition, the trend is eventually the key of predicting EOL.

Moreover, behavior of battery changes significantly after EOL. Since the main interest is to predict the number of cycles until EOL takes place, the prediction of RPT discharge capacity before EOL taking place is focused. The data of cycles before EOL are taken for the analysis and modeling process. If there is need of predicting battery behavior after EOL, another model would be suggested.

2.3 Objective

Develop a model that predict RPT cycle discharge capacity at various steps into the future before EOL.

3. Literature Review

There are many models can work with time series data. For univariate models, there are different types of regression [1] and ARIMA [2]. For multivariable model, ensemble learning [3] can be performed including decision tree, gradient boosted tree, random forest, and proposed ensemble model. Moreover, client mentions in the project proposal that fitting a square root function has been applied.

Through cycling, degradation of the lithium-ion batteries, battery type in this project, has in fact been investigated. As shown in Figure 1 in Appendix, the capacity has a relationship with respect to cycle that can be represented by linear. Furthermore, when higher and higher elevated charge voltage is applied, the relationship becomes more and more quadratic-like which is concave down, as shown in Figure 2 in Appendix. [4]

Linearity is potentially a good local or even global approximation of the relationship between capacity and number of cycles.

4. Data Cleaning and Processing

4.1 Target Variable Description

The target variable in this project is RPT discharge capacity, which is considered as an accurate measurement of retentive capacity of battery. RPT discharge capacity has a general decreasing trend with some volatility, with respect to cycle number. In addition, RPT discharge capacity is not recorded in a continuous way. Each RPT cycle takes place after a series of regular cycles (approximately 20 cycles). Sometimes consecutive RPT cycles take place, with early RPT cycles stabilizing the battery and the last RPT cycle being the actual measurement, as mentioned in the provided project proposal.

4.2 Data Cleaning and Feature Preparation

The raw data has each record corresponding to a state of charge (SOC) within a cycle. A cycle has 4 SOCs (thus 4 records) and each SOC has 3 measurements as features. The set of 4 records corresponding to same cycle are combined as one, such that each record has 12 features along with discharge capacity.

Since the target variable is RPT discharge capacity, data of all batteries that only have regular cycles are eliminated. There are 22 batteries that have RPT cycles. The remaining data have no null value. Some cycles have data for less than 4 SOC, but they belong to the batteries that have no RPT cycle. As mentioned in Section 4.1, there are some sets of consecutive RPT cycles. For these consecutive sets of RPT cycles, only the last RPT cycle of each consecutive set is retained.

The dataset is then reformatted such that each record contains the discharge capacity and 12 features for both regular and RPT as if both measurements are performed at same cycle. Since a cycle can only be either regular or RPT, this reformatting results in a lot of null values. In addition, the dataset has some cycles missing. These null values are handled by performing linear interpolation and linear extrapolation (for initial cycles), with the assumption of local linearity, which is suggested by client. Then, a continuous time series dataset is obtained, with cycle number being the time stamp.

Lastly, there are some batteries have too few RPT cycles. Some of the models require first 100 or even 110 cycles for training to forecast the later cycles. Therefore, there are 2 batteries eliminated again because they do not have RPT cycles after cycle 100.

The dataset after cleaning and processing consists of 20 batteries. The data for each battery is a set of continuous time series data. Each record has discharge capacity and 12 features for both regular and RPT cycles.

5. Model Development

5.1 Simple Moving Average (SMA) Model

SMA is the simplest model. It is univariable model. The future values are forecasted by taking mean of past values. The model can be optimized by choosing size of window of taking mean. However, there is a crucial limitation of this model: it assumes the future values are close to past values. In this scenario, the RPT discharge capacity has a negative trend. Using data of early cycles that locate at upper part in terms of magnitude would confine the predictions also inside the upper part. Depending on size of window, the prediction eventually will converge, because it is simply taking mean of past values repeatedly. A small window size of 5 is chosen, such that the model converging at value close to the last RPT cycle, as shown in Figure 3 in Appendix.

Due to its limitation, SMA model is not expected to give accurate prediction of future RPT discharge capacities. However, it can be used as a baseline model. As will be exhibited in the Model Result component, the SMAPEs of SMA are in fact not large, and growing as forecasting time horizon increases. This not only demonstrates an underlying trend of RPT discharge capacity cycle, but also implies that the rate of change is small. The later models are expected to result in smaller SMAPEs, or the learned trends of batteries differ from the actual trends by a lot.

5.2 ARIMA Model

While SMA is simply taking mean of past values and does not analyze the trend of data, ARIMA is a univariate model that allows analysis with orders of differencing, which is important detrending process for data with trend (nonstationary data). Therefore, ARIMA model is attempted after realizing the limitation of SMA. ARIMA is a complex autoregressive model that involves 3 key parameters: p, d, and q, which are lag order, degree of differencing, and order of moving average. It has been a challenge that there is no embedded ARIMA model in Spark library like some other models (linear regression, random forest, and gradient boosted tree). A piece of code is found online and studied. By tuning the three parameters using first 100 cycles of the dataset, a set of parameters is obtained that can produce model with linear negative trend (the degree of differencing is 1). The first 100 cycles are used to train ARIMA model to forecast future values for each battery. As shown in Figure 4 in Appendix, the linear negative trend produced by ARIMA model is a good approximation of the actual negative trend.

Since the actual RPT cycles are sparse and most RPT data are filled by linear interpolation, window sizes for lagging and moving average have some degree of flexibility without changing the model significantly. The main challenge of ARIMA model is to determine the order of differencing parameter, d. By attempting different values of d, it is found that only first order of differencing can make the data stationary. This also implies that the underlying trend of RPT discharge capacity is best represented by linearity, among different polynomial order. The limitation of ARIMA is that it eventually converges to a linear regression model, thus has same limitation as linear regression, as will be discussed later. In addition, ARIMA model is complicated and a piece of code online is studied and used, therefore it is hard to implement additional techniques into it. For instance, the outlier removing for linear regression.

5.3 Linear Regression Model

As first order differencing can make the data stationary when developing ARIMA model, linear regression is potentially a good model. The linear regression model is univariate and take first 110 cycles to train for a linear relationship between RPT discharge capacity and cycle number. Actual discrete RPT discharge capacity data are used for developing linear regression model instead of the continuous data after interpolation and extrapolation. In fact, the process of interpolation and extrapolation is under the assumption of local linearity. Since an RPT cycle takes place approximately every 20 cycles (can be less frequent), there are in fact very few points for training. In addition, as mentioned for SMA model, the rate of change of the negative trend is small. Therefore, any outlier due to volatility of RPT discharge capacity can impact the linear regression model significantly. With a regular linear regression that simply fits on the early RPT cycles with cycle number less than 110, the performance is poor, as showed in Table 1. The effect of outliers can be lessened to some degree by taking in more points, but a RPT cycle is expensive can require another 20 regular cycles.

The challenge of this model is to deal with outlier with such few data points. Several steps have been designed to eliminate significant outliers and make the line of linear regression model as appropriate as possible. The first step is setting upper and lower bounds for RPT discharge capacity cycles, these thresholds are very loose and just for outliers that are obviously problematic. The second step is to fit a linear regression model first and eliminate outliers based on deviation (threshold value is tuned by using two batteries as validation set) of the training data points from the line. Second step is repeated for multiple times if multiple outliers are found. However, since the data points are too few, the second point can eliminate significant outliers and might eliminate good data points when there are more than one outlier data points among the few training data points. The RPT discharge capacity measured from last RPT cycle of consecutive cycles are more reliable and less volatile than single RPT cycle, additional weights are given to training points corresponding to consecutive RPT cycles by replicating the data points twice. The third step not only help determining outliers in second step, but also lessen the effect of insignificant outliers, because those RPT discharge capacities are more assured. As showed in Table 1, the linear regression model obtained after these three steps are implemented has a much better performance.

Time Horizon	**50**	**100**	**150**	**200**	**250**	**300**	**350**
Regular Lin. Reg.	5.83	8.18	9.33	10.92	8.69	22.19	13.50
Enhanced Lin. Reg.	2.36	3.12	1.65	1.88	1.94	1.93	3.86

Table 1. Performance Comparison between Regular and Enhanced Lin. Reg. model

Linear regression model has two main limitation. First one is that it is trying to ignore the volatility of RPT discharge capacity, which is hard to accomplish with the amount of data given. Second limitation is shown in Figure 5 in Appendix, a line is indeed not a perfect fit for some batteries (in fact, it is perfect fit for some) and there is secondary term left over. Error accumulates from this secondary term as forecasting time horizon increases.

5.4 Random Forest

While the previous three models are all univariate, random forest is used as a multivariable approach. The challenge of random forest is that the data is nonstationary. When use early 100 cycles to train a random forest, the model would not recognize the later cycles because their values are all lower (except for some due to outliers) than the earlier training data. The predictions would converge to a flat line as SMA model. While ARIMA demonstrates that first order differencing can make the data stationary, the same technique is applied for random forest. However, detrending for ARIMA can be done by setting parameter d, while the detrending needs to be implemented for random forest. Random forest uses all 24 SOC features along with regular discharge capacity as features. First order differencing is applied to all features. The target variable also becomes the difference of RPT discharge capacity between two consecutive cycles. Feature selection has not been done for this model. Since first order of differencing is used, this model exhibits the limitation of linear regression.

5.5 Gradient Boosted Tree (GBT)

Challenge of GBT is also that it can only be applied to stationary time series data. Instead of detrending, GBT model turns to an approach different from previous four models. GBT model is designed to predict slope and intercept of linear model of RPT discharge capacity using only regular cycle data. Since frequency of actual regular cycle data is much higher than actual RPT cycle data, enough information can be obtained from regular cycle, with much less early cycles required.

Since the regular cycle SOC features have small variation during the cycling process of each battery, mean of each for first 50 cycles can represent the regular cycle SOC data of a battery. In addition, the regular cycle discharge capacity also has a linear-like relationship with respect to cycle number. Therefore, the slope and intercept of linear model of regular cycle discharge capacity derived from first 50 cycles are extracted as 2 features. These 14 features in total from regular cycles of data are used to train GBT for each of the two target variables: slope and intercept of linear model of RPT discharge capacity. The reformatted data has 20 records, each corresponding to a battery. Two batteries are kept as testing set and 18 batteries are split into training and validation set with ratio of 7:3. The labels (2 target variables) of validation and training set are derived by fitting linear regression to RPT discharge capacity data for each battery.

Feature selection is done with PCA. The performance on validation set is the best when using the top 5 features are output from PCA. In addition, other hyperparameters for GBT trees: maxIter, maxDepth, stepSize, are tuned. The tuning process accomplished by TrainValidationSplit with paramGrid.

Limitation of this model exhibits the limitation of linear regression model. In addition, there are not enough data for different batteries for this approach. Moreover, the outlier removing process in linear regression is not adopted for extracting features (not working well with whole set of data for a battery, which might need implementation of RANSAC).

6. Result Discussion

The table summarizing the SMAPE for 5 models with different time horizons are placed in Appendix (section 9.3). ARIMA, linear regression, GBT provide reasonably good approximation of the actual data. In specific, linear regression has the best performance, which is expected. Since other model use linear interpolation, linear extrapolation, and first order differencing for detrending, and GBT is predicting a linear model of RPT discharge capacity with regular cycle data. The models all eventually end up with a linear model and exhibit the limitation of linear regression model, which can be shown by viewing the plots in Appendix (section 9.2) Moreover, as mentioned in linear regression model section, outliers can be a big performance degradation factor when the actual RPT data points are not many. With the outlier removing steps, linear regression model can find the more appropriate line for the batteries. SMA has relatively poor performance because, as mentioned, the model does not involve the trend of data. The reason of GBT having poor performance is that it has not yet been tuned and feature selection also has not yet been done, which causing it to have extremely bad performance for some batteries. Besides SMAPE, it is important to notice that GBT requires much less cycles (50 cycles in this scenario) to make predictions into the future because it makes use of regular cycle data. Making use of regular cycle data is also an expectation from the client.

7. Future Improvement and Personal Contribution

Since the models all exhibit limitation of linear regression, it is important for dealing with limitation of linear regression. A secondary term can be added to the fitting model on top of linear equation to lessen the accumulation of error over long time horizon forecasting. Meanwhile, if more data of RPT cycles are given, the volatility of RPT cycles might be modeled, which indeed also depends on how the RPT cycles are arranged (consecutive RPT cycles give more stable measurements). In addition, if data of more batteries are given, the performance of GBT is also expected to be increased (too few batteries currently). Moreover, hyperparameter tuning and feature selection need to be completed for random forest model. Furthermore, RANSAC might be implemented for the whole data set of a battery (the outlier removing has been designed for only earlier cycles), such that models can produce more accurate approximation of the trend.

Throughout this project, I have participated in team data cleaning and processing, and done individual data processing to understand the data better. I developed SMA and linear regression model, and given suggestions to other teammates constructing their models.

8. References

[1] Ostrom, C. W. (1990). *Time Series Analysis: Regression Techniques* (Vol. 9).
[2] Krome, C., Sander, V. Time series analysis with apache spark and its applications to energy informatics. *Energy Inform* **1,** 40 (2018). https://doi.org/10.1186/s42162-018-0043-1
[3] Galicia, A., Talavera-Llames, R., Troncoso, A., Koprinska, I., & Martínez-Álvarez, F. (2018, October 12). Multi-step forecasting for big data time series based on ensemble learning. Retrieved December 22, 2020.

[4] BU-808: How to Prolong Lithium-based Batteries. (n.d.). Retrieved December 22, 2020, from https://batteryuniversity.com/learn/article/how_to_prolong_lithium_based_batteries

9. Appendix

9.1 Literature Review Plot

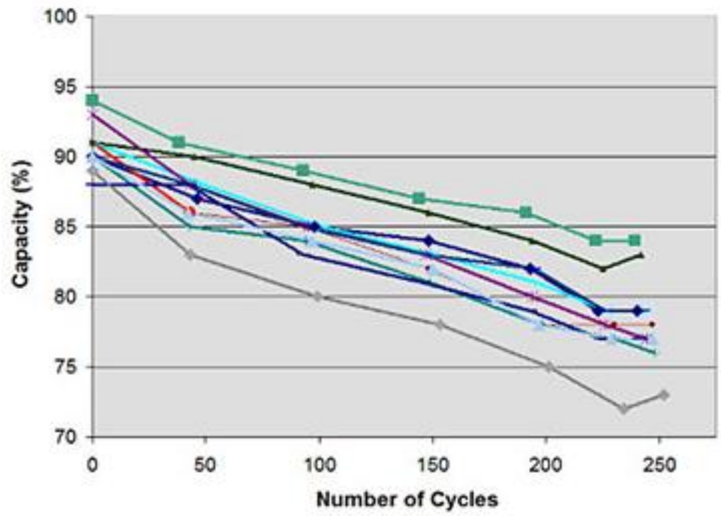

Figure 1. Ageing of Lithium-ion Battery

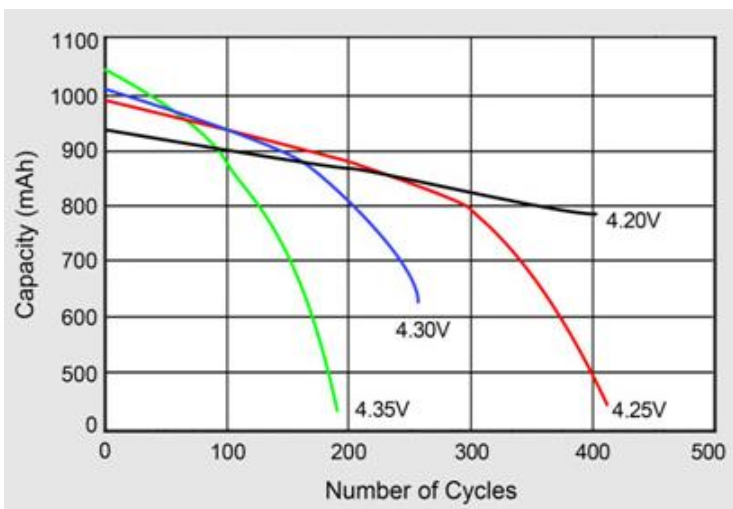

Figure 2. Ageing of Lithium-ion Battery with Elevated Charge Voltage

9.2 Comparison of Predictions and Actual Values for 5 Models

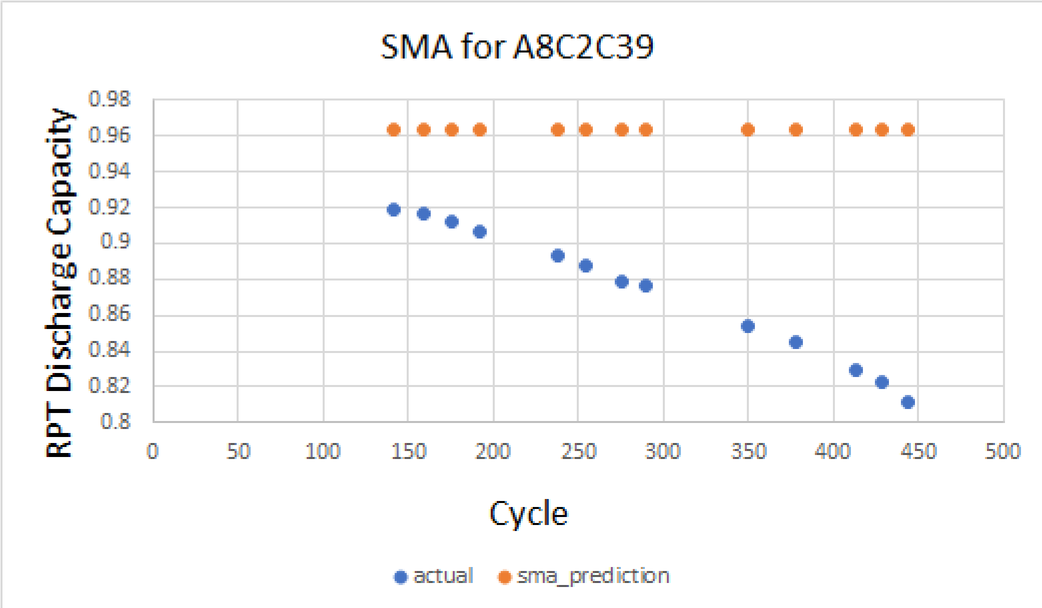

Figure 3. Predictions vs. Actual Values for SMA Model

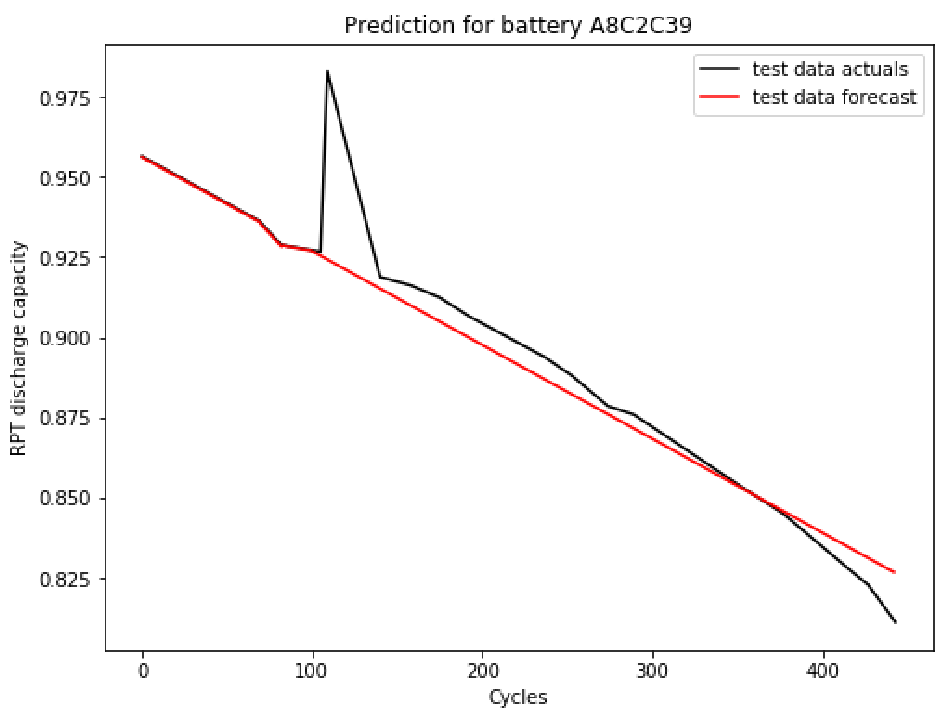

Figure 4. Predictions vs. Actual Values for ARIMA Model

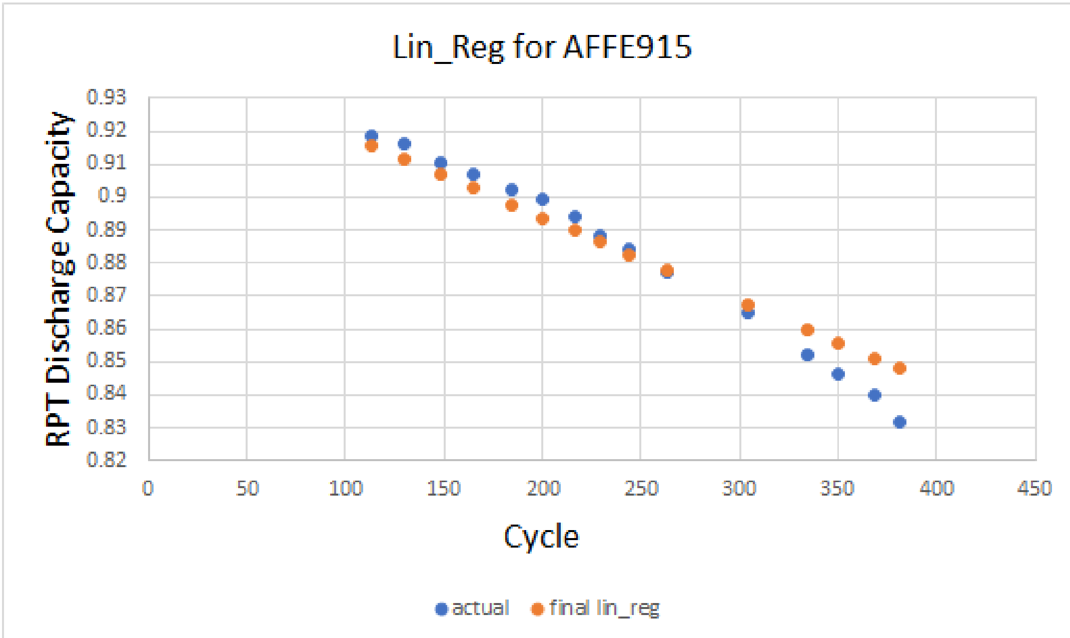

Figure 5. Predictions vs. Actual Values for Lin. Reg. Model

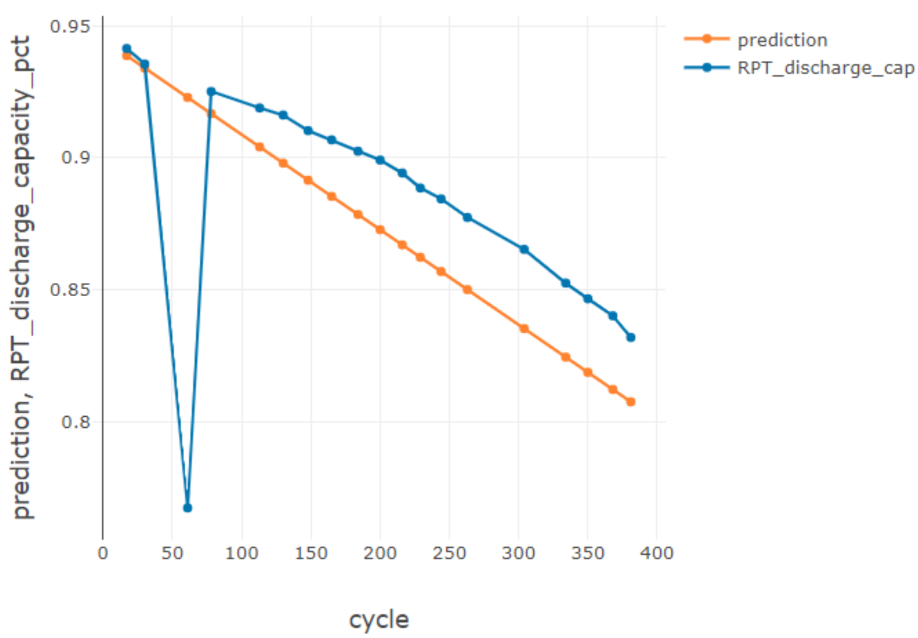

Figure 6. Predictions vs. Actual Values for Random Forest Model

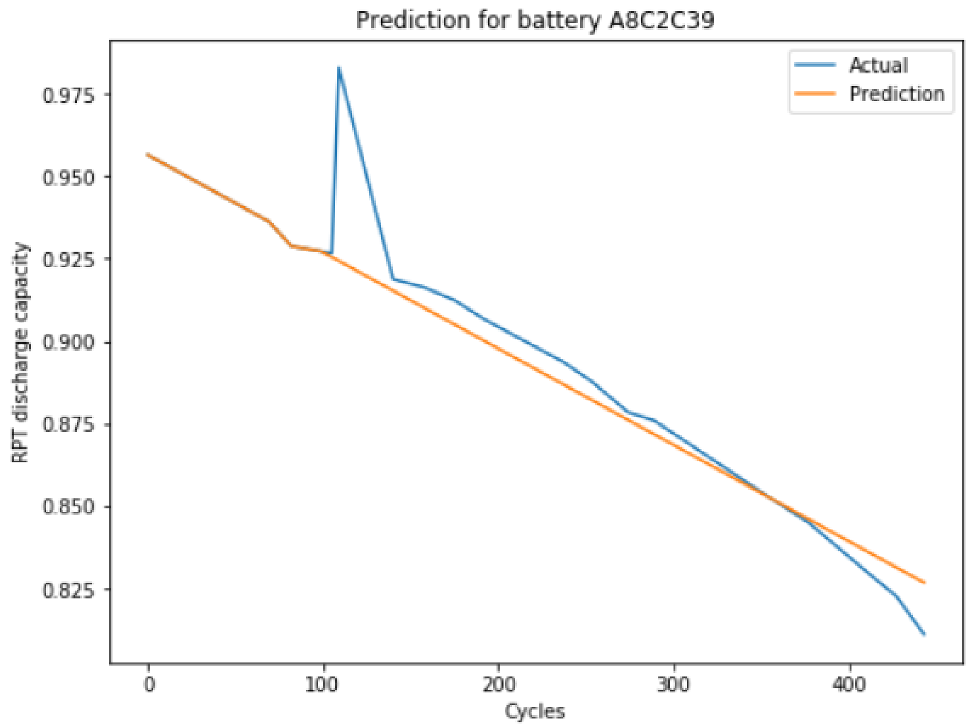

Figure 7. Predictions vs. Actual Values for GBT Model

9.3 Table of SMAPE for 5 Models

Time Horizon	**50**	**100**	**150**	**200**	**250**	**300**	**350**
SMA	3.30	4.55	6.43	7.62	8.69	10.08	12.17
ARIMA	2.43	4.51	5.55	6.92	7.21	7.39	7.39
Linear Regression	2.36	3.12	1.65	1.88	1.94	1.93	3.86
GBT	1.88	5.85	6.63	6.83	8.24	8.23	9.28
Random Forest	11.04	20.52	24.14	26.28	27.15	27.75	27.76

Table 2. SMAPE for 5 Models

9.4 Code (see following pages)

```
# -*- coding: utf-8 -*-
"""group8_notebook_clone_1.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1fwyGSZJZ-USeQvPqzeakNz0QYouXsLdA
"""

'''
MIE1628: Summative Course Project; GBatteries Team 8
Team members: Junzhe Liu (1002416557), Yue Niu (1002222613), David Wang (1001393351), Zijian Wei
    (1002276823), Yixin Xiao (1002115221)

GBatteries is an advanced battery technology company pioneering an alternative way to charge lithium-ion (Li
    -ion) batteries. Their charging method uses adaptive pulses instead of the conventional charging
    protocol  CCCV  used as the standard Li-ion charging method. Their technology enables ultra-fast charge
    without compromising the health of the battery.

The goal for this project is to develop a model to predict the RPT discharge capacity at various steps into
    the future. Labeled data is provided for thousands of cycles; this notebook will explore modeling
    solutions that will eventually play a role in a monitoring solution that will be deployed in the
    GBatteries battery testing facility. More information about the company, project requirements, and
    targets can be found in the project report (completed on a per-contributor basis).

The general steps undertaken in this notebook will be as follows:
  * Cleaning and other transformations
  * Time series visualization
  * Removing non-stationarity or seasonality (not a problem in this problem domain)
  * Partitioning and vector of past valuies
  * Train and evaluate various models; model optimization
  * Discuss results
'''

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# i. IMPORTS + LOADING DATASET
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# !pip install --upgrade pip
# !pip install fbprophet
# !pip install koalas
# !pip install missingno
# !pip install fancyimpute
# !pip install flint


from pyspark import since, SparkContext
from pyspark.sql.column import Column, _to_java_column, _to_seq
from pyspark.sql import SparkSession, Window
from pyspark.sql.functions import countDistinct
from pyspark.sql import functions as F
from pyspark.sql.types import *
from pyspark.sql.types import StructType, IntegerType, DateType, StringType, DoubleType, StructField,
    FloatType
from pyspark.sql.functions import pandas_udf, PandasUDFType, collect_list, struct, create_map
from pyspark.sql.functions import sum, max, col, avg, concat, lit, isnan, when, count, first, last
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.linalg import Vectors
from pyspark.sql import Row
from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window
from pyspark.ml.regression import GBTRegressor
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.linalg.distributed import RowMatrix
from pyspark.ml.feature import PCA
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder,TrainValidationSplit
from pyspark.ml import Pipeline
```

```python

# import warnings, sys, os, gc, time
from itertools import chain
# import missingno as msno
import numpy as np
import pandas as pd
# from fbprophet import Prophet
# from statsmodels.tsa.stattools import adfuller, acf, pacf



# load data; dbfs:/FileStore/shared_uploads/gbatteries_team8@outlook.com/dataset.csv
schema = StructType([
    StructField("_c0", IntegerType()),
    StructField("battery_model", StringType()),
    StructField("battery_id", StringType()),
    StructField("cycle", IntegerType()),
    StructField("cycle_type", StringType()),
    StructField("charge_duration_sec", IntegerType()),
    StructField("discharge_capacity_pct", FloatType()),
    StructField("soc_region", IntegerType()),
    StructField("feature_1", FloatType()),
    StructField("feature_2", FloatType()),
    StructField("feature_3", FloatType())
])

df = spark.read.format("csv").load("dbfs:/FileStore/shared_uploads/gbatteries_team8@outlook.com/dataset.csv"
    , header = True, schema = schema)
print(df.show(2))

# get schema of our data
# df.printSchema()

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 1. DATA CLEANING, PREPROCESSING
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# display count of nans; the data has been thoroughly cleaned and prepared by the GBatteries point
# of contact
print('validate that there are no null records:')
# df.select([count(when(isnan(c), c)).alias(c) for c in df.columns]).show()
df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.columns]).show()

'''
Let's move onwards to EDA and preprocessing. Seran Thirugnanam from GBatteries has outlined in the
project brief + introductory touch point meeting that the data has been thoroughly cleaned, and
has requested the project team to instead focus on improving modeling capabilities
'''

print('sample frequency of RPT vs regular cycles for a sample battery (id)')
df.filter('battery_id == "A0047B2"').select('cycle', 'cycle_type').distinct().orderBy('cycle').show(10)

'''
Note that some cycles have missing information (e.g. battery A0047B2 is missing cycle 3 in the below example
    )
'''

# show distinct (unique) values for soc_region
print('unique soc_region values')
df.select("soc_region").distinct().show()

# show distinct (unique) values for battery_model
print('unique battery models contained in dataset')
df.select("battery_model").distinct().show()

# remove records for cycles with soc less than 4
# df = df.filter(df['battery_id'] != 'A2A5795').filter((df['battery_id'] != 'ACF5ADD') | (df['cycle'] !=
    215))
```

```
# xxx = df.groupBy("battery_id").agg(countDistinct("soc_region"))
# xxx.filter(xxx[’count(soc_region)’] == 2).show()
# df.filter(df[’battery_id’] != ’ACF5ADD’).show(444)

’’’
Note that the GBatteries representative (Seran) commented on cycle types varying between charges, with RPT cycle types
being exceedingly infrequent as compared to Regular cycles. RPT is a standardized score; RPT cycles are what we care about
here, but they are measured once every ~50 cycles or so.

We can interpolate RPT metrics for every charge cycle, and Regular metrics for every charge cycle...
  * I.e. have rows for cycle 1 ... n for both RPT and Regular
  * These features can then be used for modelling (e.g. Random Forest requires populated values for every cycle)

As a first step, we also need to pivot the table so that SOC region 1...4 and their corresponding features are represented
as columns (objective: one row per battery cycle, with all features captured at a column level).

Objective:
| SOC1_f1 | SOC2_f1 | SOC3_f1 | SOC4_f1 | SOC2_f1 | SOC2_f2 | ... | SOC4_f3 |

Let’s start with pivoting our data:
’’’

# replace soc_region values ahead of pivoting
soc_dict = {1:’SOC1’, 2:’SOC2’, 3:’SOC3’, 4:’SOC4’}
mapping_expr = create_map([lit(x) for x in chain(*soc_dict.items())])
df = df.withColumn(’soc_region’, mapping_expr[df[’soc_region’]])

# pivot soc_region with all three features
dff = df.groupBy(’battery_model’, ’battery_id’, ’cycle’, ’cycle_type’, ’charge_duration_sec’, ’discharge_capacity_pct’).pivot(’soc_region’).agg(F.first(’feature_1’).alias(’feature_1’),F.first(’feature_2’).alias(’feature_2’),F.first(’feature_3’).alias(’feature_3’))

# order by battery_id and cycle (ascending)
order_cols = [’battery_id’, ’cycle’]
dff = dff.orderBy(order_cols, ascending = True)

dff.show(10)

df.show()

dff_2 = dff
window_next = Window.partitionBy(’battery_id’).orderBy(’cycle’)
dff_2 = dff_2.withColumn(’next_cycle’, F.lead(’cycle’, 1).over(window_next))
dff_2 = dff_2.withColumn(’next_cycle_type’, F.lead(’cycle_type’, 1).over(window_next))
dff_2 = dff_2.filter(dff_2[’next_cycle’].isNull()|((dff_2[’cycle_type’] == ’Regular’) | (dff_2[’next_cycle’] - dff_2[’cycle’] != 1) | (dff_2[’next_cycle_type’] != ’RPT’)))

# get dff count to compare to dff_2 (after operations)
print(dff.count())

# expected 8 less, correct
print(dff_2.count()) # > 478

dff = dff_2.drop(’next_cycle’,’next_cycle_type’)

dff_2.show(200)

dff.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("dbfs:/FileStore/uninterpolated_1219v1.csv")

dff.show()

’’’
Next, we move onto interpolating RPT AND Regular metrics for every charge cycle... i.e. have rows for
```

```python
cycle 1 ... n for both RPT and Regular cycle types
'''

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 2. INTERPOLATING ALL REGULAR + RPT CYCLES
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# Descriptive statistics for both RPT and Regular cycles
print('RPT charge_duration_sec descriptive statistics:')
dff.filter(df["cycle_type"] == "RPT").select('charge_duration_sec').describe().show()

print('Regular charge_duration_sec descriptive statistics:')
dff.filter(df["cycle_type"] == "Regular").select('charge_duration_sec').describe().show()

# we now divide our dataset into two DataFrames: one for RPT cycles and the other for Regular cycles
reg = dff.filter(df['cycle_type'] == 'Regular')
rpt = dff.filter(df['cycle_type'] == 'RPT')

#data.select([count(when(isnan(c), c)).alias(c) for c in data.columns]).show()
# the records for selected batteries all have 4 soc, this section can be commented
# get batteries
# lt_4_soc = df.groupBy("battery_id", "cycle_type", "cycle").agg(count("*"))
# lt_4_soc = lt_4_soc.filter(F.col('count(1)').isin([0,1,2,3]))
# lt4_soc_ids = [x.battery_id for x in lt_4_soc.select('battery_id').distinct().collect()]

# filter out batteries with <4 SOC_regions
# reg = reg.filter(~col('battery_id').isin(lt4_soc_ids))
# rpt = rpt.filter(~col('battery_id').isin(lt4_soc_ids))

'''
Because we're interested in forecasting out RPT cycles, it doesn't make sense to
analyze batteries (battery_id) where we have no RPT cycle values. The operation
below will exclude batteries that don't have any RPT cycle records.
'''
# the batteries we selected have RPT cycles, this part is commented
# get DataFrame of unique batteries that DO have RPT cycle records
unique_rpt_ids = rpt.groupBy("battery_id").agg(countDistinct("cycle"))

# apply a filter on the Regular cycles DataFrame to exclude only the
# aforementioned batteries
rpt_ids_arr = [str(row['battery_id']) for row in unique_rpt_ids.collect()]
reg = reg.filter(F.col('battery_id').isin(rpt_ids_arr))

# compare to confirm we have the same count of battery ids in both DataFrames
unique_reg_ids = reg.groupBy("battery_id").agg(countDistinct("cycle"))
print('# reg batteries:', unique_reg_ids.count(), '# rpt batteries:', unique_rpt_ids.count())
battery_id_list = [str(i) for i in np.array(unique_reg_ids.collect())[:,0]]

'''
The sample data generally consists of batteries undergoing a few hundred cycles of charge-discharge (max cycle = 563).
We need cycles 1 ... n_i for each battery i; to do this, we construct new DataFrames for Regular and RPT cycles, then
interpolate/fill NaN values (i.e. rows in the new DataFrame corresponding to missing cycles in the original DataFrame).

Note here that the dataset exclusively looks at battery_model == 'A'; we can exclude this column since it conveys no
information.
'''
#reg.groupBy('battery_id').agg(F.max('cycle')).show()
#rpt.groupBy('battery_id').agg(F.max('cycle')).show()

# get max cycle counts per battery
max_cycle_counts = dff.groupBy('battery_id').agg(F.max('cycle'))
max_cycle_counts = max_cycle_counts.filter(col("battery_id").isin(battery_id_list))
max_counts_arr = np.array(max_cycle_counts.collect())

```

```
# initialize lists we will use to construct our new DataFrame (new DataFrame to account for missing cycles)
id_arr = []
cyc_arr = []

# create lists to initialize "all cycle" DataFrames with
for id, count in max_counts_arr:
    for cycle_no in range(1, int(count) + 1):
        id_arr.append(str(id))
        cyc_arr.append(cycle_no)

# create full DataFrame with all cycle information (f => stands for full)
f_reg = sqlContext.createDataFrame(zip(id_arr, cyc_arr), schema=['battery_id', 'cycle'])
f_rpt = sqlContext.createDataFrame(zip(id_arr, cyc_arr), schema=['battery_id', 'cycle'])

# populate existing data; we now have
f_reg = f_reg.join(reg, on = ['battery_id', 'cycle'], how = 'left')
f_rpt = f_rpt.join(rpt, on = ['battery_id', 'cycle'], how = 'left')

# sort; easier on the eyes
f_reg = f_reg.orderBy(order_cols, ascending = True)
f_rpt = f_rpt.orderBy(order_cols, ascending = True)

# display results
# print('Regular cycle DataFrame with missing values:')
# f_reg.show(5)
# print('RPT cycle DataFrame with missing values:')
# f_rpt.show(5)

# print missingness of RPT cycle DataFrame (recall from before, RPT cycles are far and many between; these
# are only measure every ~50 cycles according to Seran)
print('RPT cycle DataFrame missingness pct:', (1 - rpt.count()/f_rpt.count())*100,'%')

# drop battery_model, which are all 'A'
f_reg = f_reg.drop('battery_model')
f_rpt = f_rpt.drop('battery_model')

'''
There aren't any out-of-the-box solutions for advanced imputation (e.g. MICE, KNN) solutions in PySpark.
At the recommendation of GBatteries, we'll look at the following for both f_rpt (RPT cycles) and f_reg
(Regular cycles):

1. Linear interpolation to fill in NULL discharge_capacity_pct values
2. Forward filling in NULL charge_duration_sec - i.e. propagating the last valid observation forward
(rationale: the charge_duration_sec values reasonable similar within all RPT cycle records, and within
all Regular cycle records).

'''

def coalesce(*cols):
    """
    Returns the first non-null argument if exists. Otherwise, null.
    source: https://spark.apache.org/docs/2.1.0/api/python/_modules/pyspark/sql/functions.html

    :param *cols: a variable number of columns

    :returns: returns the first column that is not null.
    """
    sc = SparkContext._active_spark_context
    jc = sc._jvm.functions.coalesce(_to_seq(sc, cols, _to_java_column))
    return Column(jc)

# linear interpolation window transformation to impute discharge_capacity_pct
def fill_linear_interpolation(df, id_cols, order_col, value_col):
    """
    Apply linear interpolation to dataframe to fill gaps.

    :param df: Spark DataFrame
    :param id_cols: string or list of column names to partition by the window function
```

```
    :param order_col: column to use to order by the window function
    :param value_col: column to be filled

    :returns: spark dataframe updated with interpolated values
    """
    # create row number over window and a column with row number only for non missing values
    w = Window.partitionBy(id_cols).orderBy(order_col)
    new_df = df.withColumn('rn',F.row_number().over(w))
    new_df = new_df.withColumn('rn_not_null',F.when(F.col(value_col).isNotNull(),F.col('rn')))

    # create relative references to the start value (last value not missing)
    w_start = Window.partitionBy(id_cols).orderBy(order_col).rowsBetween(Window.unboundedPreceding,-1)
    new_df = new_df.withColumn('start_val',F.last(value_col,True).over(w_start))
    new_df = new_df.withColumn('start_rn',F.last('rn_not_null',True).over(w_start))

    # create relative references to the end value (first value not missing)
    w_end = Window.partitionBy(id_cols).orderBy(order_col).rowsBetween(0,Window.unboundedFollowing)
    new_df = new_df.withColumn('end_val',F.first(value_col,True).over(w_end))
    new_df = new_df.withColumn('end_rn',F.first('rn_not_null',True).over(w_end))

    # create references to gap length and current gap position
    new_df = new_df.withColumn('diff_rn',F.col('end_rn')-F.col('start_rn'))
    new_df = new_df.withColumn('curr_rn',F.col('diff_rn')-(F.col('end_rn')-F.col('rn')))

    # calculate linear interpolation value
    lin_interp_func = (F.col('start_val')+(F.col('end_val')-F.col('start_val'))/F.col('diff_rn')*F.col('curr_rn'))
    new_df = new_df.withColumn(value_col,F.when(F.col(value_col).isNull(),lin_interp_func).otherwise(F.col(value_col)))

    if not isinstance(id_cols, list):
        id_cols = [id_cols]

    keep_cols = id_cols + [order_col, value_col]
    new_df = new_df.select(keep_cols)
    return new_df


# forward fill (ffill) interpolation window transformation to impute charge_duration_sec
def forward_fill_interpolation(df, id_col, value_col):
    """
    Apply forward fill to interpolate missing values in dataframe.

    :param df: Spark DataFrame
    :param id_cols: string column name to partition by the window function
    :param value_col: column to be filled

    :returns: spark dataframe updated with interpolated values
    """

    w1 = Window.partitionBy(id_col).rowsBetween(Window.unboundedPreceding, Window.currentRow)
    w2 = Window.partitionBy(id_col).rowsBetween(Window.currentRow, Window.unboundedFollowing)

    new_df = df.withColumn('previous', last(value_col, ignorenulls = True).over(w1))\
      .withColumn('next', first(value_col, ignorenulls = True).over(w2))\
      .withColumn('new_score', (coalesce(F.col('previous'), F.col('next') + coalesce(F.col('next'), F.col('previous')) / 2)))\
      .drop('next', 'previous')

    new_df = new_df.select('battery_id', 'cycle', 'new_score')
    new_df = new_df.withColumnRenamed('new_score', 'charge_duration_sec')

    return new_df

# pipeline that we can apply to both the RPT and Regular cycle DataFrames
def imputation_pipeline(df, target_feature, ffill_feature, soc_features):
    """
    Apply imputation functions (linear, forward fill) to interpolate
```

```python
    missing values in our Regular cycle and RPT cycle DataFrames.

    :param df: input Spark DataFrame
    :param target_feature: target; linear interpolation
    :param ffill_feature: feature; forward fill interpolation
    :param soc_features: list of other features to be linearly interpolated

    :returns: full spark dataframe updated with interpolated values
    """

    # interpolate
    cap_pct = fill_linear_interpolation(df = df, id_cols = 'battery_id', order_col = 'cycle', value_col = 'discharge_capacity_pct')

    # interpolate
    chg_dur = forward_fill_interpolation(df = df, id_col = 'battery_id', value_col = 'charge_duration_sec')

    # construct output DataFrame
    new_df = cap_pct.join(chg_dur, on = ['battery_id', 'cycle'], how = 'left')

    # interpolate SOC region features
    for soc_f in soc_features:
        intermediate = fill_linear_interpolation(df = df, id_cols = 'battery_id', order_col = 'cycle', value_col = soc_f)
        new_df = new_df.join(intermediate, on = ['battery_id', 'cycle'], how = 'left')

    return new_df

# features/target to be imputed
soc_features = ['SOC1_feature_1', 'SOC1_feature_2', 'SOC1_feature_3',
                'SOC2_feature_1', 'SOC2_feature_2', 'SOC2_feature_3',
                'SOC3_feature_1', 'SOC3_feature_2', 'SOC3_feature_3',
                'SOC4_feature_1', 'SOC4_feature_2', 'SOC4_feature_3']
target_feature = 'discharge_capacity_pct'
ffill_feature = 'charge_duration_sec'

# new DataFrame for RPT cycles; all NULL values have been imputed, as detailed above (i => stands for interpolated)
# then sort by battery_id and cycle
i_rpt = imputation_pipeline(f_rpt, target_feature = 'discharge_capacity_pct', ffill_feature = 'charge_duration_sec', soc_features = soc_features)
i_rpt = i_rpt.orderBy(order_cols, ascending = True)
i_rpt.cache()

# new DataFrame for Regular cycles; all NULL values have been imputed, as detailed above (i => stands for interpolated)
# then sort by battery_id and cycle
i_reg = imputation_pipeline(f_reg, target_feature = 'discharge_capacity_pct', ffill_feature = 'charge_duration_sec', soc_features = soc_features)
i_reg = i_reg.orderBy(order_cols, ascending = True)
i_reg.cache()

'''
Now that we've generated full datasets (containing records for all cycles) for both RPT + Regular cycles
(note again we've aligned on this approach with the GBatteries team), we can generate a unified dataset by
joining the Regular DataFrame to the RPT dataset (on 'battery_id' and 'cycle').

Regular cycle features provide valuable information that may improve model performance (source: GBatteries team).
We'll accomplish this by:

1. Applying column name labels (prepending) on our features to distinguish between RPT and Regular cycle features.
2. Performing a join operation on 'battery_id' and 'cycle'.

Then, we'll move into our modeling phase.
'''

```

```python
# prepend feature names with 'REG_' and 'RPT_'
old_feature_names = ['discharge_capacity_pct', 'charge_duration_sec'] + soc_features
rpt_feature_names = ['RPT_' + old_name for old_name in old_feature_names]
reg_feature_names = ['REG_' + old_name for old_name in old_feature_names]

# create column name mapping to apply to i_rpt and i_reg
new_rpt_dict = dict(zip(old_feature_names, rpt_feature_names))
new_reg_dict = dict(zip(old_feature_names, reg_feature_names))

def rename_columns(df, columns):
    if isinstance(columns, dict):
        for old_name, new_name in columns.items():
            df = df.withColumnRenamed(old_name, new_name)
        return df
    else:
        raise ValueError("'columns' should be a dict, like {'old_name_1':'new_name_1', 'old_name_2':'
    new_name_2'}")

# prepend column names with 'REG_' and 'RPT_', respectively
i_reg = rename_columns(i_reg, new_reg_dict)
i_rpt = rename_columns(i_rpt, new_rpt_dict)

# get final resultant DataFrame
data = i_rpt.join(i_reg, on = ['battery_id', 'cycle'], how = 'left')
data = data.orderBy(order_cols, ascending = True)

# data.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("dbfs:/FileStore/
    unextrapolated_1219v1.csv")

data.count()

# data.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("dbfs:/FileStore/
    unextrapolated_1219v1.csv")

'''
Note here that our linear interpolation function only captures the 'inside' of
two bounding values. It does not interpolate in the case that the first or last
records for a battery_id are NULL. We need to create a custom function to
address this.

Once this is done, we'll have our final DataFrame 'data' that we can start modeling
with. We'll save this as a temp view/table (for the purposes of providing code to
the client, as a csv file), which we'll then read as our model input.
'''

# linear interpolation window transformation to impute discharge_capacity_pct
def beginning_extrapolation(df, id_cols, order_col, value_col):
    """
    Apply linear interpolation to dataframe to fill gaps.

    :param df: Spark DataFrame
    :param id_cols: string or list of column names to partition by the window function
    :param order_col: column to use to order by the window function
    :param value_col: column to be filled

    :returns: spark dataframe updated with interpolated values
    """
    # create row number over window and a column with row number only for non missing values
    lead_window = Window.partitionBy(id_cols).orderBy(order_col)
    df_2 = df.withColumn("next_value", F.when(F.col(value_col).isNotNull(),F.lead(value_col, 1).over(
    lead_window)))

    w = Window.partitionBy(id_cols).orderBy(order_col)
    new_df = df_2.withColumn('rn',F.row_number().over(w))
    new_df = new_df.withColumn('rn_not_null',F.when(F.col(value_col).isNotNull(),F.col('rn')))

    # create relative references to the end values (first 2 value not missing)
    w_end = Window.partitionBy(id_cols).orderBy(order_col).rowsBetween(0,Window.unboundedFollowing)
```

```python
    new_df = new_df.withColumn('end_val',F.first(value_col,True).over(w_end))
    new_df = new_df.withColumn('end_rn',F.first('rn_not_null',True).over(w_end))
    new_df = new_df.withColumn('end_val_2',F.first('next_value',True).over(w_end))

    # create references to gap length and current gap position
    new_df = new_df.withColumn('diff_rn',F.col('end_rn')-F.col('rn'))

    # calculate linear interpolation value
    lin_interp_func = (F.col('end_val')-(F.col('end_val_2')-F.col('end_val'))*F.col('diff_rn'))
    new_df = new_df.withColumn(value_col,F.when(F.col(value_col).isNull(),lin_interp_func).otherwise(F.col(
     value_col)))

    if not isinstance(id_cols, list):
        id_cols = [id_cols]

    new_df = new_df.drop('next_value','rn','rn_not_null','end_val','end_rn','end_val_2','diff_rn')
    return new_df

def ending_extrapolation(df, id_cols, order_col, value_col):
    """
    Apply linear interpolation to dataframe to fill gaps.

    :param df: Spark DataFrame
    :param id_cols: string or list of column names to partition by the window function
    :param order_col: column to use to order by the window function
    :param value_col: column to be filled

    :returns: spark dataframe updated with interpolated values
    """
    # create row number over window and a column with row number only for non missing values
    lag_window = Window.partitionBy(id_cols).orderBy(order_col)
    df_2 = df.withColumn("last_value",  F.when(F.col(value_col).isNotNull(),F.lag(value_col, 1).over(
     lag_window)))

    w = Window.partitionBy(id_cols).orderBy(order_col)
    new_df = df_2.withColumn('rn',F.row_number().over(w))
    new_df = new_df.withColumn('rn_not_null',F.when(F.col(value_col).isNotNull(),F.col('rn')))

    # create relative references to the start value (last value not missing)
    w_start = Window.partitionBy(id_cols).orderBy(order_col).rowsBetween(Window.unboundedPreceding,-1)
    new_df = new_df.withColumn('start_val',F.last(value_col,True).over(w_start))
    new_df = new_df.withColumn('start_rn',F.last('rn_not_null',True).over(w_start))
    new_df = new_df.withColumn('start_val_2',F.first('last_value',True).over(w_start))

    # create references to gap length and current gap position
    new_df = new_df.withColumn('diff_rn',F.col('rn')-F.col('start_rn'))

    # calculate linear interpolation value
    lin_interp_func = (F.col('start_val')+(F.col('start_val')-F.col('start_val_2'))*F.col('diff_rn'))
    new_df = new_df.withColumn(value_col,F.when(F.col(value_col).isNull(),lin_interp_func).otherwise(F.col(
     value_col)))

    if not isinstance(id_cols, list):
        id_cols = [id_cols]

    new_df = new_df.drop('last_value','rn','rn_not_null','start_val','start_rn','start_val_2','diff_rn')
    return new_df

# get final df
df_extrapolated = data

# apply changes per the two functions above
for col_name in ['RPT_discharge_capacity_pct','RPT_SOC1_feature_1','RPT_SOC1_feature_2','RPT_SOC1_feature_3'
     ,'RPT_SOC2_feature_1','RPT_SOC2_feature_2','RPT_SOC2_feature_3','RPT_SOC3_feature_1','RPT_SOC3_feature_2
     ','RPT_SOC3_feature_3','RPT_SOC4_feature_1','RPT_SOC4_feature_2','RPT_SOC4_feature_3','
     REG_discharge_capacity_pct','REG_SOC1_feature_1','REG_SOC1_feature_2','REG_SOC1_feature_3','
     REG_SOC2_feature_1','REG_SOC2_feature_2','REG_SOC2_feature_3','REG_SOC3_feature_1','REG_SOC3_feature_2',
     'REG_SOC3_feature_3','REG_SOC4_feature_1','REG_SOC4_feature_2','REG_SOC4_feature_3']:
```

```
    df_extrapolated = beginning_extrapolation(df = df_extrapolated, id_cols = 'battery_id', order_col = 'cycle', value_col = col_name)

# save as csv (alternatively, save as view ... but we're working with Databricks on a client project; we'll
# produce the cleaned dataset as a .csv file for easier handoff to the GBatteries team)
df_extrapolated = ending_extrapolation(df = df_extrapolated, id_cols = 'battery_id', order_col = 'cycle', value_col = col_name)

df_extrapolated.cache()
df_extrapolated.count()

# save as csv
df_extrapolated.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("dbfs:/FileStore/extrapolated_output_v7.csv")

# select only the data of each battery before end of life. Battery A0047B2 has last RPT cycle at 96, too few data, and battery A35D29D has only 2 RPT cycles before reaching eol. These 2 batteries are not used for modeling.
df_eol = df_extrapolated.filter((col("battery_id") != "A8C2C39")|(col("cycle") < 445)).filter((col("battery_id") != "A65D895")|(col("cycle") < 368)).filter((col("battery_id") != "A2C0E4A")|(col("cycle") < 329)).filter((col("battery_id") != "AFFE915")|(col("cycle") < 382)).filter((col("battery_id") != "A947A53")|(col("cycle") < 284)).filter((col("battery_id") != "A1D35B0")|(col("cycle") < 280)).filter((col("battery_id") != "A74A962")|(col("cycle") < 182)).filter((col("battery_id") != "AFD5A8F")|(col("cycle") < 226)).filter((col("battery_id") != "AEBE15E")|(col("cycle") < 138)).filter((col("battery_id") != "A716092")|(col("cycle") < 213)).filter((col("battery_id") != "A18242C")|(col("cycle") < 220)).filter((col("battery_id") != "A4E5D13")|(col("cycle") < 240)).filter((col("battery_id") != "AEE8EF6")|(col("cycle") < 247)).filter((col("battery_id") != "A231712")|(col("cycle") < 209)).filter((col("battery_id") != "A6AD931")|(col("cycle") < 231)).filter(col("battery_id") != "A35D29D").filter((col("battery_id") != "AC3C95D")|(col("cycle") < 119)).filter((col("battery_id") != "A83B987")|(col("cycle") < 211)).filter((col("battery_id") != "A73CF24")|(col("cycle") < 209)).filter((col("battery_id") != "A0047B2")|(col("cycle") < 127)).filter((col("battery_id") != "AD5B491")|(col("cycle") < 185)).filter((col("battery_id") != "A43046B"))

df_eol.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("dbfs:/FileStore/cleaned_data.csv")

'''
Before we move onto the modeling phase. We'll take a look at feature importance. Here we'll
also read in the temp view/table/csv that we saved above. To avoid running all the data processing
steps again.

Here, our cleaned dataset contains only two batteries -- we've disregarded the other battery cycles with
lower cycle counts (we cannot evaluate sMAPE on higher cycle counts if the raw data does not contain this
information). GBatteries has supported this approach (selecting a small subset of batteries to analyze). Since
are selecting batteries that are representative of the dataset, our methodology can easily be applied to any
of the other batteries in the dataset as well. Citation: GBatteries representative (see meeting notes)
'''

# load cleaned data; dbfs:/FileStore/shared_uploads/gbatteries_team8@outlook.com/extrapolated_v6.csv
schema_cleaned = StructType([
    StructField("battery_id", StringType()), StructField("cycle", IntegerType()),
    StructField("RPT_discharge_capacity_pct", FloatType()), StructField("RPT_charge_duration_sec", FloatType()),
    StructField("RPT_SOC1_feature_1", FloatType()), StructField("RPT_SOC1_feature_2", FloatType()), StructField("RPT_SOC1_feature_3", FloatType()),
    StructField("RPT_SOC2_feature_1", FloatType()), StructField("RPT_SOC2_feature_2", FloatType()), StructField("RPT_SOC2_feature_3", FloatType()),
    StructField("RPT_SOC3_feature_1", FloatType()), StructField("RPT_SOC3_feature_2", FloatType()), StructField("RPT_SOC3_feature_3", FloatType()),
    StructField("RPT_SOC4_feature_1", FloatType()), StructField("RPT_SOC4_feature_2", FloatType()), StructField("RPT_SOC4_feature_3", FloatType()),
    StructField("REG_discharge_capacity_pct", FloatType()), StructField("REG_charge_duration_sec", FloatType()),
    StructField("REG_SOC1_feature_1", FloatType()), StructField("REG_SOC1_feature_2", FloatType()), StructField("REG_SOC1_feature_3", FloatType()),
    StructField("REG_SOC2_feature_1", FloatType()), StructField("REG_SOC2_feature_2", FloatType()), StructField("REG_SOC2_feature_3", FloatType()),
```

```
    StructField("REG_SOC3_feature_1", FloatType()), StructField("REG_SOC3_feature_2", FloatType()), StructField("REG_SOC3_feature_3", FloatType()),
    StructField("REG_SOC4_feature_1", FloatType()), StructField("REG_SOC4_feature_2", FloatType()), StructField("REG_SOC4_feature_3", FloatType())
])

# cleaned dataset for 2 batteries that we will model ...
dff = spark.read.format("csv").load("dbfs:/FileStore/shared_uploads/gbatteries_team8@outlook.com/cleaned_data.csv", header = True, schema = schema_cleaned)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 2.1 SEASONALITY AND STATIONARY
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

'''
Now we'll move onto the modeling phase. In this section, we'll test several (>=5) time series models
on our processed data set and compare results. Specifically, we'll touch upon the following for EACH
model...

  * Notes on what affects performance and sMAPE for each model
  * Basic model information and how the model was built
  * Model construction and optimization process
  * Challenges and limitations

We'll cover the following 5 models in this notebook...
  * Linear Regression
  * Simple Moving Average
  * ARIMA
  * Random Forest
  * XGBoost

'''

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 3. MODELING, MODEL EVALUATION (5 MODELS)
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

'''
The RPT discharge capacity has a linear-like relationship with cycle number and linearity is an
important concept throughout the project (e.g. linear interpolation, linear extrapolation, and
first order differencing). Linear regression is definitely a model needed to be attempted.
While performance of regualr linear regression suffers from outliers, enhanced linear regression has
outlier rejection algorithm involved.
'''

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 3.a) LINEAR REGRESSION
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
schema_cleaned = StructType([
    StructField("battery_id", StringType()), StructField("cycle", IntegerType()),
    StructField("discharge_capacity_pct", DoubleType()), StructField("consecutive_rpt", IntegerType())
])

data = spark.read.format("csv").load("dbfs:/FileStore/tables/rpt_actual_with_consec.csv",header = True, schema = schema_cleaned)

# get battery id list
unique_id = data.select("battery_id").distinct()
unique_id_list = [str(i) for i in np.array(unique_id.collect())[:,0]]

# create empty result dataframe
schema = StructType([
  StructField('battery_id', StringType(), True),
  StructField('cycle', IntegerType(), True),
  StructField('discharge_capacity_pct', DoubleType(), True),
  StructField('regular_prediction',IntegerType(),True),
  StructField('enhanced_prediction',IntegerType(),True)
  ])
```

```python
df_result = spark.createDataFrame([], schema)

# data dataframe setup for applying ml
assembler = VectorAssembler(inputCols=["cycle"],outputCol="features")
data_ml = assembler.transform(data)
data_ml = data_ml.withColumnRenamed("discharge_capacity_pct",'label')

# set up window
max_window = Window.partitionBy(col("battery_id")).orderBy("battery_id")
max_diff = max(col("difference")).over(max_window)

for battery_id in unique_id_list:
  # regular linear regression
  lr = LinearRegression()
  train = data_ml.filter((data_ml["battery_id"]==battery_id) & (data_ml["cycle"]<=110))
  lr_model = lr.fit(train)
  test = lr_model.transform(data_ml.filter((data_ml["battery_id"]==battery_id) & (data_ml["cycle"]>110))).withColumnRenamed("prediction","regular_prediction")

  # regression with weighted data points and outlier removing
  train = train.filter((col("label") < lit(0.95)) & (col("label") > lit(0.80)))
  train_consec = train.filter(col("consecutive_rpt") == 1)
  train = train.union(train_consec).union(train_consec)
  for i in range(3):
    lr = LinearRegression()
    lr_model_1 = lr.fit(train)
    train = lr_model_1.transform(train).withColumn("difference",F.abs(col("label") - col("prediction")))
    count_1 = train.count()
    train = train.select(col("*"), max_diff.alias("max_difference")).filter((col("difference") != col("max_difference")) | (col("max_difference") < lit(0.045))).drop("difference","max_difference","prediction")
    count_2 = train.count()
    # no more outlier
    if (count_1 == count_2):
      break

  lr = LinearRegression()
  lr_enhanced_model = lr.fit(train)
  test = lr_enhanced_model.transform(test).withColumnRenamed("prediction","enhanced_prediction").drop("consecutive_rpt","features").withColumnRenamed("label","discharge_capacity_pct")
  df_result = df_result.union(test)

  battery_id = unique_id_list[0]

# regular linear regression
lr = LinearRegression()
train = data_ml.filter((data_ml["battery_id"]==battery_id) & (data_ml["cycle"]<=110))
lr_model = lr.fit(train)
test = lr_model.transform(data_ml.filter((data_ml["battery_id"]==battery_id) & (data_ml["cycle"]>110))).withColumnRenamed("prediction","regular_prediction")

# regression with weighted data points and outlier removing
train = train.filter((col("label") < lit(0.95)) & (col("label") > lit(0.80)))
train_consec = train.filter(col("consecutive_rpt") == 1)
train = train.union(train_consec).union(train_consec)
for i in range(3):
  lr = LinearRegression()
  lr_model_1 = lr.fit(train)
  train = lr_model_1.transform(train).withColumn("difference",F.abs(col("label") - col("prediction")))
  count_1 = train.count()
  train = train.select(col("*"), max_diff.alias("max_difference")).filter((col("difference") != col("max_difference")) | (col("max_difference") < lit(0.045))).drop("difference","max_difference","prediction")
  count_2 = train.count()
  # no more outlier
  if (count_1 == count_2):
    break

```

```python
lr = LinearRegression()
lr_enhanced_model = lr.fit(train)
test = lr_enhanced_model.transform(test).withColumnRenamed("prediction","enhanced_prediction").drop("consecutive_rpt","features").withColumnRenamed("label","discharge_capacity_pct")

# export result table as .csv
df_result.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("dbfs:/FileStore/lin_reg_result_table.csv")

# read in resultant csv just exported
res_schema = StructType([
            StructField("battery_id", StringType()), StructField("cycle", IntegerType()),
            StructField("discharge_capacity_pct", DoubleType()), StructField("regular_prediction", DoubleType()),
            StructField("enhanced_prediction", DoubleType())
])

lin_reg_res = spark.read.format("csv").load("dbfs:/FileStore/lin_reg_result_table.csv",header = True,schema = res_schema)

# compute smape
lin_reg_res = lin_reg_res.withColumn("time_step",F.when(col("cycle") < 161, 50).when(col("cycle") < 211, 100).when(col("cycle") < 211, 100).when(col("cycle") < 261, 150).when(col("cycle") < 311, 200).when(col("cycle") < 361, 250).when(col("cycle") < 411, 300).when(col("cycle") < 461, 350).otherwise(0))
lin_reg_res = lin_reg_res.withColumn("regular_expre",F.abs(col("discharge_capacity_pct")-col("regular_prediction")) * 200 / (col("discharge_capacity_pct")+col("regular_prediction")))
lin_reg_res = lin_reg_res.withColumn("enhanced_expre",F.abs(col("discharge_capacity_pct")-col("enhanced_prediction")) * 200 / (col("discharge_capacity_pct")+col("enhanced_prediction")))
enhanced_smape = lin_reg_res.groupBy("time_step").agg(F.mean("enhanced_expre").alias("smape"))
regular_smape = lin_reg_res.groupBy("time_step").agg(F.mean("regular_expre").alias("smape"))

'''
Simple Moving Average is a method of time series smoothing and is a very basic forecasting technique.
It does not need estimation of parameters, but rather is based on order selection. This will serve as
a good baseline model and highlight certain areas to address with improved, more complex models.

Specifically in this problem domain, SMA models give a lot of weight to old data -- this is something
that exponential moving averages (EMA) can address. SMAs and EMAs are used in similar ways: to identify
trends and find potential areas of 'support' or 'resistance'.
'''

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 3.b) Simple Moving Average
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

schema_cleaned = StructType([
    StructField("battery_id", StringType()), StructField("cycle", IntegerType()),
    StructField("RPT_discharge_capacity_pct", FloatType()), StructField("RPT_charge_duration_sec", FloatType()),
    StructField("RPT_SOC1_feature_1", FloatType()), StructField("RPT_SOC1_feature_2", FloatType()), StructField("RPT_SOC1_feature_3", FloatType()),
    StructField("RPT_SOC2_feature_1", FloatType()), StructField("RPT_SOC2_feature_2", FloatType()), StructField("RPT_SOC2_feature_3", FloatType()),
    StructField("RPT_SOC3_feature_1", FloatType()), StructField("RPT_SOC3_feature_2", FloatType()), StructField("RPT_SOC3_feature_3", FloatType()),
    StructField("RPT_SOC4_feature_1", FloatType()), StructField("RPT_SOC4_feature_2", FloatType()), StructField("RPT_SOC4_feature_3", FloatType()),
    StructField("REG_discharge_capacity_pct", FloatType()), StructField("REG_charge_duration_sec", FloatType()),
    StructField("REG_SOC1_feature_1", FloatType()), StructField("REG_SOC1_feature_2", FloatType()), StructField("REG_SOC1_feature_3", FloatType()),
    StructField("REG_SOC2_feature_1", FloatType()), StructField("REG_SOC2_feature_2", FloatType()), StructField("REG_SOC2_feature_3", FloatType()),
    StructField("REG_SOC3_feature_1", FloatType()), StructField("REG_SOC3_feature_2", FloatType()), StructField("REG_SOC3_feature_3", FloatType()),
    StructField("REG_SOC4_feature_1", FloatType()), StructField("REG_SOC4_feature_2", FloatType()), StructField("REG_SOC4_feature_3", FloatType())
])
```

```
data = spark.read.format("csv").load("dbfs:/FileStore/tables/final_data.csv",header = True,schema =
    schema_cleaned)

sma_data = data.select([col("battery_id"),col("cycle"),col("RPT_discharge_capacity_pct").alias("label")])
# get battery id list
unique_id = sma_data.select("battery_id").distinct()
unique_id_list = [str(i) for i in np.array(unique_id.collect())[:,0]]

# create empty result dataframe
schema = StructType([
  StructField('battery_id', StringType(), True),
  StructField('cycle', IntegerType(), True),
  StructField('sma_prediction',IntegerType(),True)
  ])
df_result = spark.createDataFrame([], schema)

# this section of code computes predictions forward with window size of 5, but takes much less time (run
    time of several seconds)
for id in unique_id_list:
  sma_data_id = sma_data.filter(col("battery_id") == id).drop("battery_id")
  sma_data_id_count = sma_data_id.count()
  sma_data_id_fill0 = sma_data_id.withColumn("label",F.when(F.col("cycle") < 101,col("label")).otherwise(lit
    (0)))
  sma_predicted_id = sma_data_id_fill0.collect()
  for i in range(101,sma_data_id_count+1):
    new_row = Row(cycle = i, label = (sma_predicted_id[i-2][1] + sma_predicted_id[i-3][1] + sma_predicted_id
    [i-4][1] + sma_predicted_id[i-5][1] + sma_predicted_id[i-6][1])/5)
    sma_predicted_id[i-1] = new_row
  sma_data_id_forecast = spark.createDataFrame(sma_predicted_id).withColumn("battery_id",lit(id)).select("
    battery_id","cycle","label").withColumnRenamed("label","sma_prediction")
  df_result = df_result.union(sma_data_id_forecast

  schema = StructType([
    StructField("battery_id", StringType()), StructField("cycle", IntegerType()),
    StructField("discharge_capacity_pct", DoubleType()), StructField("consecutive_rpt", IntegerType())
])

# this section of code also computes predictions forward, takes long time to run (not for running purpose,
    or variable name needs to be adjusted)
# sma_data_1_forecast = sma_data_1.filter(col("cycle") < 111)
# sma_data_2_forecast = sma_data_2.filter(col("cycle") < 79)
# create a one-row dataframe with cycle and predicted_label then union to the actual data
# for i in range(111,sma_data_1_count+1):
# sma_data_1_append = sma_data_1_forecast.filter(col("cycle") > (i-1)-5).drop("cycle").agg(avg(col("label"))
    .alias("label")).withColumn("cycle",lit(i)).select("cycle","label")
# sma_data_1_forecast = sma_data_1_forecast.union(sma_data_1_append).orderBy("cycle")

# for i in range(79,sma_data_2_count+1):
# sma_data_2_append = sma_data_2_forecast.filter(col("cycle") > (i-1)-5).drop("cycle").agg(avg(col("label"))
    .alias("label")).withColumn("cycle",lit(i)).select("cycle","label")
# sma_data_2_forecast = sma_data_2_forecast.union(sma_data_2_append).orderBy("cycle")


actual_data = spark.read.format("csv").load("dbfs:/FileStore/tables/rpt_actual_with_consec.csv",header =
    True,schema = schema).filter(col("cycle") > 100)

df_result = actual_data.join(df_result,["battery_id","cycle"],"left")

df_result.coalesce(1).write.format("com.databricks.spark.csv").option("header", "true").save("dbfs:/
    FileStore/sma_result_table.csv")

# read in stored resultant data from sma
schema = StructType([
    StructField("battery_id", StringType()), StructField("cycle", IntegerType()),
    StructField("discharge_capacity_pct", DoubleType()), StructField("consecutive_rpt", IntegerType()),
    StructField("sma_prediction", DoubleType())
])
```

```
sma_res = spark.read.format("csv").load("dbfs:/FileStore/sma_result_table.csv",header = True,schema = schema)

# compute smape
sma_res = sma_res.select("battery_id","cycle","discharge_capacity_pct","sma_prediction")
sma_res = sma_res.withColumn("time_step",F.when(col("cycle") < 151, 50).when(col("cycle") < 201, 100).when(col("cycle") < 211, 100).when(col("cycle") < 251, 150).when(col("cycle") < 301, 200).when(col("cycle") < 351, 250).when(col("cycle") < 401, 300).when(col("cycle") < 451, 350).otherwise(0))
sma_res = sma_res.withColumn("sma_expre",F.abs(col("discharge_capacity_pct")-col("sma_prediction")) * 200 / (col("discharge_capacity_pct")+col("sma_prediction")))
sma_smape = sma_res.groupBy("time_step").agg(F.mean("sma_expre").alias("smape"))

import matplotlib.pylab as plt

fig, ax = plt.subplots(figsize=(20,5))
ax.plot(actl, 'black', label='test data actuals')
ax.plot(pred, 'r', label='test data forecast')
#ax.plot(ts_log_diff, 'blue', label='training data')
legend = ax.legend(loc='upper left')
legend.get_frame().set_facecolor('w')
display(fig.figure)

schema_gt_ARIMA = StructType([StructField("battery_id", StringType()), StructField("cycle", IntegerType()), StructField("discharge_capacity_pct", FloatType())])
gt = spark.read.format("csv").load("dbfs:/FileStore/tables/rpt_actual_v8.csv",header = True,schema = schema_gt_ARIMA)

def sMAPE_ARIMA (pred, battery_id, num):
  ground_truth_df = gt.filter(col("battery_id")==battery_id)
  ground_truth = np.array([ground_truth_df.select("discharge_capacity_pct").collect()[i]["discharge_capacity_pct"] for i in range(ground_truth_df.filter(col("cycle")<=100).count(), ground_truth_df.filter(col("cycle")<=(100+num)).count())])
  predictions = []
  cycles = np.array([ground_truth_df.select("cycle").collect()[i]["cycle"] for i in range(ground_truth_df.filter(col("cycle")<=100).count(), ground_truth_df.filter(col("cycle")<=(100+num)).count())])

  for i in cycles:
    index = int(i)
    predictions.append(pred[index-1])
  predictions = np.array(predictions)

  return 100*np.sum(np.abs(ground_truth-predictions)/((np.abs(ground_truth)+np.abs(predictions))/2))/len(ground_truth)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 3.c) ARIMA
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

!pip install pmdarima
from pmdarima import auto_arima
from statsmodels.tsa.arima_model import ARIMA
import matplotlib.pylab as plt
import math

def ARIMA_model(battery_id):

  # Filter out one battery from the cleaned_data.csv
  train = dff.filter(col("battery_id") == battery_id).select('battery_id','cycle','RPT_discharge_capacity_pct')
  train_pandas = train.toPandas()

  # Create test array and sort in descending order, also define a sub-array to include only the first 50 datapoints
  train_array = np.array(train_pandas.RPT_discharge_capacity_pct)
  train_array100 = train_array[:100]

  # Use auto_arima to tune the parameters p,d,q for the ARIMA model
```

```
  stepwise_fit = auto_arima(train_array100,
                            start_p = 1,
                            start_q = 1,
                            max_p = 3,
                            max_q = 3,
                            m = 1,
                            seasonal = None,
                            d = 1,
                            trace = False,
                            random = True,
                            error_action ='ignore',
                            suppress_warnings = True,
                            stepwise = True)

  # Use the sub-array to fit the model, the best parameters are set according to the result from auto_ARIMA
  model = ARIMA(train_array100, order=stepwise_fit.order)
  results_ARIMA = model.fit()

  # Apply the training set to forecast the rest of the test set
  pred  = results_ARIMA.predict(1,len(train_array)-1,typ='levels')

  # Plot the actual test result and the prediction result
  fig, ax = plt.subplots(figsize=(8,6))
  ax.set(title='Prediction for battery'+str(battery_id), xlabel='Cycles', ylabel='RPT discharge capacity')
  ax.plot(train_array, 'black', label='test data actuals')
  ax.plot(pred, 'r', label='test data forecast')
  legend = ax.legend(loc='upper right')
  legend.get_frame().set_facecolor('w')
  display(fig.figure)

  # sMAPE calculations
  cycles_list = [50,100,150,200,250,300,350]
  smape_val = []
  for cycles in cycles_list:
    smape_val.append(sMAPE_ARIMA(pred,battery_id,cycles))
  smape_val = [float(val) for val in smape_val]
  sqlContext.createDataFrame(zip(cycles_list, smape_val), schema=['no_cycles(prediction)', 'smape']).show()
  smape_val = np.array(smape_val)
  return smape_val

# 20 batteries
battery_id = np.array(['A0047B2','A18242C','A1D35B0','A231712','A2C0E4A','A4E5D13','A65D895','A6AD931','
     A716092','A73CF24','A74A962','A83B987',
  'A8C2C39','A947A53','AC3C95D','AD5B491','AEBE15E','AEE8EF6','AFD5A8F','AFFE915'])

# For each battery, calculate its sMAPE
smape_sum_ARIMA = np.zeros(7)
for i in battery_id:
  a = ARIMA_model(i)
  if not (np.isnan(a[3])) :
    smape_sum_ARIMA = smape_sum_ARIMA + a

# Obtain the average sMAPE for all 20 batteries
smape_avg_ARIMA = smape_sum_ARIMA/len(battery_id)
print(smape_avg_ARIMA)

'''
Random Forest is a popular and effective ensemble machine learning algorithm,  widely used for
classification and regression predictive modeling problems. It can also be used for time series
forecasting, provided that our data is transformed using a sliding-window representation
(source: https://machinelearningmastery.com/random-forest-for-time-series-forecasting/).

Multivariate Time Series Forecasting Using Random Forest

'''

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 3.d) RANDOM FOREST
```

```python
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import RandomForestRegressor
import matplotlib.pyplot as plt
from pyspark.sql.window import Window
import pyspark.sql.functions as func
from pyspark.sql.functions import lit
from pyspark.sql.functions import expr



# Load the data file
dff_RF = spark.read.format("csv").load("dbfs:/FileStore/tables/extrapolated_v8.csv", header = True, schema = schema_cleaned)

# a function used to convert features to dense vectors
def transData(data):
    return data.rdd.map(lambda r: [Vectors.dense(r[1]),r[-1]]).toDF(['features','label'])

def RF (batt_id):
  data1 = dff_RF.filter(col("battery_id") == batt_id ).withColumn('user',lit('group8'))
  #data2 is for generating test data later on
  data2 = dff_RF.filter(col("battery_id") == batt_id ).select(['battery_id','cycle','RPT_SOC1_feature_1','RPT_SOC1_feature_2','RPT_SOC1_feature_3','RPT_SOC2_feature_1','RPT_SOC2_feature_2','RPT_SOC2_feature_3','RPT_SOC3_feature_1','RPT_SOC3_feature_2','RPT_SOC3_feature_3','RPT_SOC4_feature_1','RPT_SOC4_feature_2','RPT_SOC4_feature_3','REG_discharge_capacity_pct','REG_SOC1_feature_1','REG_SOC1_feature_2','REG_SOC1_feature_3','REG_SOC2_feature_1','REG_SOC2_feature_2','REG_SOC2_feature_3','REG_SOC3_feature_1','REG_SOC3_feature_2','REG_SOC3_feature_3','REG_SOC4_feature_1','REG_SOC4_feature_2','REG_SOC4_feature_3','RPT_discharge_capacity_pct'])

  # generate the new label
  data1 = data1.withColumn('RPT_discharge_capacity_pct_lag', func.lag(data1['RPT_discharge_capacity_pct']).over(Window.orderBy("user")))
  data1 = data1.withColumn('Difference', data1['RPT_discharge_capacity_pct'] - data1['RPT_discharge_capacity_pct_lag'] )

  # generate the new set of features
  for i in ('RPT_SOC1_feature_1','RPT_SOC1_feature_2','RPT_SOC1_feature_3','RPT_SOC2_feature_1','RPT_SOC2_feature_2','RPT_SOC2_feature_3','RPT_SOC3_feature_1','RPT_SOC3_feature_2','RPT_SOC3_feature_3','RPT_SOC4_feature_1','RPT_SOC4_feature_2','RPT_SOC4_feature_3','REG_discharge_capacity_pct','REG_SOC1_feature_1','REG_SOC1_feature_2','REG_SOC1_feature_3','REG_SOC2_feature_1','REG_SOC2_feature_2','REG_SOC2_feature_3','REG_SOC3_feature_1','REG_SOC3_feature_2','REG_SOC3_feature_3','REG_SOC4_feature_1','REG_SOC4_feature_2','REG_SOC4_feature_3'):
      data1 = data1.withColumn(f"lag_{i}", func.lag(data1[i]).over(Window.orderBy("user")))
      data1 = data1.withColumn(f"difference_{i}", data1[i] - data1[f"lag_{i}"])


  # create a new df consisting of the new features and labels
  data = data1.select(['battery_id','cycle','difference_RPT_SOC1_feature_1','difference_RPT_SOC1_feature_2','difference_RPT_SOC1_feature_3','difference_RPT_SOC2_feature_1','difference_RPT_SOC2_feature_2','difference_RPT_SOC2_feature_3','difference_RPT_SOC3_feature_1','difference_RPT_SOC3_feature_2','difference_RPT_SOC3_feature_3','difference_RPT_SOC4_feature_1','difference_RPT_SOC4_feature_2','difference_RPT_SOC4_feature_3','difference_REG_discharge_capacity_pct','difference_REG_SOC1_feature_1','difference_REG_SOC1_feature_2','difference_REG_SOC1_feature_3','difference_REG_SOC2_feature_1','difference_REG_SOC2_feature_2','difference_REG_SOC2_feature_3','difference_REG_SOC3_feature_1','difference_REG_SOC3_feature_2','difference_REG_SOC3_feature_3','difference_REG_SOC4_feature_1','difference_REG_SOC4_feature_2','difference_REG_SOC4_feature_3','difference'])
  # Convert the data to dense vector

  transformed = transData(data)
```

```
transformed2 = transData(data2)

# Deal with categorical variables
featureIndexer = VectorIndexer(inputCol="features", \
                               outputCol="indexedFeatures",\
                               maxCategories=4).fit(transformed)

data_transformed = featureIndexer.transform(transformed)
data_transformed2 = featureIndexer.transform(transformed2)
w = Window().partitionBy(lit('a')).orderBy(lit('a'))

data_transformed = data_transformed.withColumn("row_num", row_number().over(w))
data_transformed2 = data_transformed2.withColumn("row_num", row_number().over(w))

# train-test split, the first 100 cycles are used for training.
# The average feature decreasing rates are called testData, but it is used for prediction
trainingData = data_transformed.filter(col("row_num").between(2,100))
testData = (data_transformed2.filter(col('row_num') == 100).subtract(data_transformed2.filter(col('row_num
   ') == 1))).toPandas()
testData = testData.div(100)
testData = sqlContext.createDataFrame(testData)

# Define RandomForest algorithm
rf = RandomForestRegressor()

# Pipeline Architecture
pipeline = Pipeline(stages=[featureIndexer, rf])


model= pipeline.fit(trainingData)
predictions = model.transform(testData)

#set the decrease rate for the label after the first 100 training sycles
num_rows_needed = data1.count()-100
result_rows = predictions.select('prediction').withColumn('n',lit(num_rows_needed))
result_rows = result_rows.withColumn('n', expr('explode(array_repeat(n,int(n)))'))
result_rows = result_rows.drop('n')
first_row = data1.selectExpr("RPT_discharge_capacity_pct as prediction")

#use cumSum to calculate the future capacities
w = Window().partitionBy(lit('a')).orderBy(lit('a'))
first_row = first_row.withColumn("row_num", row_number().over(w))
first_row = first_row.filter(col("row_num").between(100,100))
first_row = first_row.drop('row_num')
result_rows = first_row.union(result_rows)
result_rows = result_rows.withColumn("row_num",row_number().over(w))
result_rows = result_rows.withColumn('RPT_prediction', sum('prediction').over(Window.orderBy('row_num')))
result_rows = result_rows.select('RPT_prediction')
training_rows = data1.selectExpr("RPT_discharge_capacity_pct as RPT_prediction").limit(99)
result = training_rows.union(result_rows)




# Plot
pred = np.array([val.RPT_prediction for val in result.select('RPT_prediction').collect()])
actual = np.array([val.RPT_discharge_capacity_pct for val in data1.select('RPT_discharge_capacity_pct').
   collect()])

fig, ax = plt.subplots(figsize=(8,6))
ax.set(title='Prediction for battery ' + str(batt_id), xlabel='Cycles', ylabel='RPT discharge capacity')
ax.plot(actual, label='Actual')
ax.plot(pred, label='Prediction')
legend = ax.legend(loc='upper right')
display(fig.figure)

# compute sMAPE
num_forcast = [50,100,150,200,250,300,350]
```

```
  smape_val = []
  for num in num_forcast:
    smape_val.append(sMAPE_RF(pred,batt_id,num))
  smape_val = [float(val) for val in smape_val]
  sqlContext.createDataFrame(zip(num_forcast, smape_val), schema=['no_cycles(prediction)', 'smape']).show()
  smape_val = np.array(smape_val)
  return smape_val

# import ground truth
schema_gt_rf = StructType([StructField("battery_id", StringType()), StructField("cycle", IntegerType()), StructField("discharge_capacity_pct", FloatType())])
gt = spark.read.format("csv").load("dbfs:/FileStore/tables/rpt_actual_v8.csv",header = True,schema = schema_gt_rf)

#calculate sMAPE
def sMAPE_RF (pred, batt_id, num):
  ground_truth_df = gt.filter(col("battery_id")==batt_id)
  ground_truth = np.array([ground_truth_df.select("discharge_capacity_pct").collect()[i]["discharge_capacity_pct"] for i in range(ground_truth_df.filter(col("cycle")<=100).count(), ground_truth_df.filter(col("cycle")<=(100+num)).count())])
  predictions = []
  cycles = np.array([ground_truth_df.select("cycle").collect()[i]["cycle"] for i in range(ground_truth_df.filter(col("cycle")<=100).count(), ground_truth_df.filter(col("cycle")<=(100+num)).count())])

  for i in cycles:
    index = int(i)
    predictions.append(pred[index-1])
  predictions = np.array(predictions)

  return 100*np.sum(np.abs(ground_truth-predictions)/((np.abs(ground_truth)+np.abs(predictions))/2))/len(ground_truth)

#RF calculate average for all batteries
batt_id = np.array(dff_RF.select("battery_id").distinct().collect())
smape_sum = np.zeros(7)
b = 0
for i in batt_id:
  #two of the abandoned batteries are still in the df, removed
  if i[0] != 'A43046B' and i[0] != 'A35D29D':
    a = RF(i[0])
    if not (np.isnan(a[3])) :
      smape_sum = smape_sum + a
      b = b+1
smape_avg = smape_sum/b

batt_id = np.array(['AEBE15E', 'A716092', 'A4E5D13', 'A83B987', 'AEE8EF6', 'A8C2C39', 'AC3C95D', 'A6AD931', 'A18242C', 'A1D35B0', 'AFD5A8F', 'A947A53', 'A74A962'])
smape_sum = np.zeros(7)
for i in batt_id:
  a = RF(i)
  if not (np.isnan(a[3])) :
    smape_sum = smape_sum + a
smape_avg = smape_sum/len(batt_id)

# get the avg smape
print(smape_avg)

'''
GBT
'''

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 3.e) GBT
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
schema_cleaned = StructType([
    StructField("battery_id", StringType()), StructField("cycle", IntegerType()),
    StructField("RPT_discharge_capacity_pct", FloatType()), StructField("RPT_charge_duration_sec", FloatType()),
```

```
    StructField("RPT_SOC1_feature_1", FloatType()), StructField("RPT_SOC1_feature_2", FloatType()),
     StructField("RPT_SOC1_feature_3", FloatType()),
    StructField("RPT_SOC2_feature_1", FloatType()), StructField("RPT_SOC2_feature_2", FloatType()),
     StructField("RPT_SOC2_feature_3", FloatType()),
    StructField("RPT_SOC3_feature_1", FloatType()), StructField("RPT_SOC3_feature_2", FloatType()),
     StructField("RPT_SOC3_feature_3", FloatType()),
    StructField("RPT_SOC4_feature_1", FloatType()), StructField("RPT_SOC4_feature_2", FloatType()),
     StructField("RPT_SOC4_feature_3", FloatType()),
    StructField("REG_discharge_capacity_pct", FloatType()), StructField("REG_charge_duration_sec", FloatType
     ()),
    StructField("REG_SOC1_feature_1", FloatType()), StructField("REG_SOC1_feature_2", FloatType()),
     StructField("REG_SOC1_feature_3", FloatType()),
    StructField("REG_SOC2_feature_1", FloatType()), StructField("REG_SOC2_feature_2", FloatType()),
     StructField("REG_SOC2_feature_3", FloatType()),
    StructField("REG_SOC3_feature_1", FloatType()), StructField("REG_SOC3_feature_2", FloatType()),
     StructField("REG_SOC3_feature_3", FloatType()),
    StructField("REG_SOC4_feature_1", FloatType()), StructField("REG_SOC4_feature_2", FloatType()),
     StructField("REG_SOC4_feature_3", FloatType())
])

schema_gt = StructType([StructField("battery_model", StringType()), StructField("battery_id", StringType()),
      StructField("cycle", IntegerType()),StructField("RPT_discharge_capacity_pct", FloatType())])

df = spark.read.format("csv").load("dbfs:/FileStore/tables/extrapolated_v6-1.csv", header = True, schema =
     schema_cleaned)
gt = spark.read.format("csv").load("dbfs:/FileStore/tables/cleaned_data_RPT-2.csv", header = True, schema =
     schema_gt).filter((col("battery_id")=="A8C2C39") | (col("battery_id")=="AFFE915")).drop("battery_model")


# list of battery and their eol
bad_batt = ["AEBE15E","AEE8EF6","A231712","A35D29D","AC3C95D","AD5B491","A43046B"]
bad_eol = [254,249,238,179,191,196,221]

battery = ["A716092", "A4E5D13", "A83B987",  "A73CF24", "A6AD931", "A18242C", "A1D35B0", "AFD5A8F", "A947A53
     ", "A74A962", "A2C0E4A", "A0047B2", "A65D895","A8C2C39","AFFE915"]
eol = [212,239,210,208,230,219,298,225,375,181,328,146,403,444,381]

all_battery = battery + bad_batt
all_eol = eol + bad_eol


# STEP 1 Linear Regression of both Regular and RPT Batteries
def LR(df, battery_list, eol_list):
  assembler = VectorAssembler(
    inputCols=["cycle"],
    outputCol="features")

  lr = LinearRegression()

  coef_rpt = []
  inter_rpt =[]
  coef_reg = []
  inter_reg =[]

  for i in range(len(battery_list)):
    train = df.filter((df["battery_id"]==battery_list[i]) & (df["cycle"]<=eol_list[i])).withColumnRenamed("
     RPT_discharge_capacity_pct", "label").select("cycle","label")
    a = assembler.transform(train)
    lrModel = lr.fit(a)
    coef_rpt.append(lrModel.coefficients.values[0].item())
    inter_rpt.append(lrModel.intercept)

  for i in range(len(battery_list)):
    train = df.filter((df["battery_id"]==battery_list[i]) & (df["cycle"]<=eol_list[i])).withColumnRenamed("
     REG_discharge_capacity_pct", "label").select("cycle","label")
    a = assembler.transform(train)
    lrModel = lr.fit(a)
    coef_reg.append(lrModel.coefficients.values[0].item())
```

```
    inter_reg.append(lrModel.intercept)

  return coef_rpt, inter_rpt, coef_reg, inter_reg


# Step 2 Prepare data for GBT
def dataPrepare_gbt(df, battery_list, coef_rpt, inter_rpt, coef_reg, inter_reg, threshold=50):
  w = Window().orderBy(lit('A'))

  feature = df.columns[18:] # only regular features

  mean_df = df.filter((df["cycle"]<=threshold) & (df["battery_id"].isin(battery_list))).select(feature+["battery_id"]).groupBy("battery_id").agg(*[avg(c) for c in feature]).withColumn("row_num", row_number().over(w))

  c_rpt_df = sc.parallelize(coef_rpt).map(Row("coef_rpt")).toDF().withColumn("row_num", row_number().over(w))
  i_rpt_df = sc.parallelize(inter_rpt).map(Row("inter_rpt")).toDF().withColumn("row_num", row_number().over(w))
  c_reg_df = sc.parallelize(coef_reg).map(Row("coef_reg")).toDF().withColumn("row_num", row_number().over(w))
  i_reg_df = sc.parallelize(inter_reg).map(Row("inter_reg")).toDF().withColumn("row_num", row_number().over(w))

  final_df = c_rpt_df.join(i_rpt_df, c_rpt_df.row_num == i_rpt_df.row_num)\
                     .join(c_reg_df, c_rpt_df.row_num == c_reg_df.row_num)\
                     .join(i_reg_df, c_rpt_df.row_num == i_reg_df.row_num)\
                     .join(mean_df, c_rpt_df.row_num == mean_df.row_num).drop("row_num")

  return final_df


# Step 3a GBT & Hyperparameter tuning
def GBT(df, label = 'coef_rpt'):
  # assembler
  feature = df.columns[2:4] + df.columns[5:]
  assembler = VectorAssembler(
              inputCols= feature,
              outputCol="features")
  #gbt_input = assembler_gbt.transform(df) 0
  #features_df = gbt_input.select("features")
  #pca = PCA(k = num_feature, inputCol="features", outputCol="pca_features")
  #pca_model = pca.fit(features_df)
  #gbt_df = pca_model.transform(gbt_input)
  pca = PCA(inputCol="features", outputCol="pca_features")

  # data split
  trainSet = df.filter((col("battery_id")!="A8C2C39") & (col("battery_id")!="AFFE915"))
  testSet = df.filter((col("battery_id")=="A8C2C39") | (col("battery_id")=="AFFE915"))

  # gbt
  gbt = GBTRegressor(featuresCol = 'pca_features', labelCol = label)
  #gbt_model = gbt.fit(trainSet)

  pipeline = Pipeline(stages=[assembler,pca,gbt])

  # parameter grid
  paramGrid = ParamGridBuilder() \
            .addGrid(pca.k, [5, 8, 10, 14])\
            .addGrid(gbt.maxIter, [3, 5, 10]) \
            .addGrid(gbt.maxDepth, [3, 6, 9]) \
            .addGrid(gbt.stepSize, [0.01, 0.05, 0.1]) \
            .build()

  # evaluator
  evaluator_reg = RegressionEvaluator(labelCol=label, predictionCol="prediction", metricName="rmse")
```

```
  #train-val splid for parameter tuning
  tvs = TrainValidationSplit(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator_reg,
                          trainRatio = 0.7)

  # train the model
  tvsModel = tvs.fit(trainSet)

  # prediction
  gbt_predictions_train = tvsModel.transform(trainSet)
  gbt_predictions_test = tvsModel.transform(testSet)

  #loss
  rmse_train = evaluator_reg.evaluate(gbt_predictions_train)
  rmse_test = evaluator_reg.evaluate(gbt_predictions_test)

  print("Train Loss: {}\nTest Loss: {}\n".format(rmse_train,rmse_test))

  return tvsModel, gbt_predictions_train, gbt_predictions_test



# Step 3b GBT after Hyperparameter tuning
def GBT_after(df, label, num_feature,ite,depth,step):
  # assembler
  feature = df.columns[2:4] + df.columns[5:]
  assembler_gbt = VectorAssembler(
              inputCols= feature,
              outputCol="features")
  gbt_input = assembler_gbt.transform(df)

  # feature reduction
  features_df = gbt_input.select("features")
  pca = PCA(k = num_feature, inputCol="features", outputCol="pca_features")
  pca_model = pca.fit(features_df)
  gbt_df = pca_model.transform(gbt_input)

  # data split
  trainSet = gbt_df.filter((col("battery_id")!="A8C2C39") & (col("battery_id")!="AFFE915"))
  testSet = gbt_df.filter((col("battery_id")=="A8C2C39") | (col("battery_id")=="AFFE915"))

  # gbt
  gbt = GBTRegressor(featuresCol = 'pca_features', labelCol = label, maxIter=ite,maxDepth=depth,stepSize=
    step)
  gbt_model = gbt.fit(trainSet)

  # evaluator
  evaluator_reg = RegressionEvaluator(labelCol=label, predictionCol="prediction", metricName="rmse")

  # prediction
  gbt_predictions_train = gbt_model.transform(trainSet)
  gbt_predictions_test = gbt_model.transform(testSet)

  #loss
  rmse_train = evaluator_reg.evaluate(gbt_predictions_train)
  rmse_test = evaluator_reg.evaluate(gbt_predictions_test)

  print("Train Loss: {}\nTest Loss: {}\n".format(rmse_train,rmse_test))

  return gbt_model, gbt_predictions_train, gbt_predictions_test



# Step 4 calculate sMAPE
def sMAPE_calculation(pred_coef_df, pred_inter_df, battery_id, num):
  coef_test = pred_coef_df.select("battery_id","prediction").withColumnRenamed("prediction","pred_coef")
  inter_test = pred_inter_df.select("battery_id","prediction").withColumnRenamed("prediction","pred_inter")
```

```
  coef = gbt_pred_test_coef.filter((col("battery_id")==battery_id)).select("prediction").collect()[0]["
    prediction"]
  inter = gbt_pred_test_inter.filter((col("battery_id")==battery_id)).select("prediction").collect()[0]["
    prediction"]

  compare = gt.filter(col("battery_id")==battery_id).withColumn("prediction",col("cycle")*coef+inter)

  a = np.array([compare.select("RPT_discharge_capacity_pct").collect()[i]["RPT_discharge_capacity_pct"] for
    i in range(compare.filter((col("cycle")>50) & (col("cycle")<(50+num))).count())])
  b = np.array([compare.select("prediction").collect()[i]["prediction"] for i in range(compare.filter((col("
    cycle")>50) & (col("cycle")<(50+num))).count())])

  sMAPE = np.sum(np.abs(a-b)/((np.abs(a)+np.abs(b))/2))/len(a)

  return compare, sMAPE



# Implementation
# Linear Regression to get coefficient and constant for each battery id
# coef_rpt, inter_rpt, coef_reg, inter_reg = LR(df, battery, eol)
coef_rpt, inter_rpt, coef_reg, inter_reg = LR(df, all_battery, all_eol)

# prepare date for gbt
threshold = 100
lr_df = dataPrepare_gbt(df, all_battery, coef_rpt, inter_rpt, coef_reg, inter_reg, threshold)

# gbt & hyperparameter tuning  # running time = 30 mins
#model_coef, train_pred_coef, test_pred_coef = GBT(lr_df, 'coef_rpt')
##threshold=50: numFeature = 14, maxIter = 3, maxDepth = 6, stepSize = 0.01
##threshold=100: numFeature = 14, maxIter = 10, maxDepth = 3, stepSize = 0.1

#model_inter, train_pred_inter, test_pred_inter = GBT(lr_df, 'inter_rpt')
##threshold=50: numFeature = 5, maxIter = 10, maxDepth = 6, stepSize = 0.1
##threshold=100: numFeature = 8, maxIter = 10, maxDepth = 3, stepSize = 0.1


# use GBT_after() with the best combination of hyperparameters for different threshold or different number
    of forecast
gbt_model_coef, gbt_pred_train_coef, gbt_pred_test_coef = GBT_after(lr_df, 'coef_rpt', 14,  10, 3, 0.1)
gbt_model_inter, gbt_pred_train_inter, gbt_pred_test_inter = GBT_after(lr_df, 'inter_rpt', 8,  10, 3, 0.1)

# sMAPE
num_forcast = [50,100,150,200,250,300,350,400]
for num in num_forcast:
  com_df1, sMAPE_1 = sMAPE_calculation(gbt_pred_test_coef, gbt_pred_test_inter, "A8C2C39",num)
  com_df2, sMAPE_2 = sMAPE_calculation(gbt_pred_test_coef, gbt_pred_test_inter, "AFFE915",num)
  print("number of forcast: {}, A8C2C39 sMAPE: {}, AFFE915 sMAPE: {}, Average sMAPE: {}".format(num, sMAPE_1
    , sMAPE_2,(sMAPE_1+sMAPE_2)/2))



# display(gbt_pred_test_coef.select("battery_id","coef_rpt","prediction"))
# display(gbt_pred_test_inter.select("battery_id","inter_rpt","prediction"))
# display(com_df1.select("cycle","RPT_discharge_capacity_pct","prediction"))
# display(com_df2.select("cycle","RPT_discharge_capacity_pct","prediction"))

# Important
# This section is some Scala code utilizing school cluster for this project, not for running purpose because
     it does not run on databricks
# Include data cleaning and processing processes with some similar as the main script in pyspark and some
    different (side individual analysis)
# Linear regression tuning is also included
// the output data set has no null or NaN, and each record represents a cycle 12 features (3 features from
    each soc)
// only cycles with 4 soc regions are included, 37 records from raw_data are removed
import org.apache.spark.sql.Row
```

```scala
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.sql.Row
import org.apache.spark.ml.feature.VectorAssembler

// the given data set from client
val df_raw = spark.table("gbatteries_team_8.raw_data")

val df_1 = df_raw.groupBy("battery_id","cycle").agg(sum("soc_region") as "soc_region_sum", count(lit(1)) as "soc_region_count")
val df_2 = df_1.filter($"soc_region_sum" =!= 10 || $"soc_region_count" =!= 4)

// soc_region_sum
df_1.groupBy("soc_region_sum","soc_region_count").agg(count(lit(1))).show()
df_2.groupBy("soc_region_sum","soc_region_count").agg(count(lit(1))).show()

df_2.orderBy("battery_id","cycle").show(df_2.count().asInstanceOf[Int],false)

// no null or NaN values
val df_3 = df_raw.na.drop("any") // drop all records with any entry being null or NaN
df_3.count()

df_raw.select("battery_id").distinct().count() // 48 batteries
val df_cleaned = df_raw.filter($"battery_id" =!= "A2A5795").filter($"battery_id" =!= "ACF5ADD" || $"cycle" =!= 215)

// df_cleaned has no null or NaN, and all cycles have 4 soc
df_cleaned.count() // 31760
df_raw.count() // 32266
df_raw.filter($"battery_id" === "A2A5795").count() // 503
df_raw.filter($"battery_id" === "ACF5ADD" && $"cycle" === 215).count() // 3

// check if all batteries have RPT cycles
df_cleaned.select("battery_id").distinct().count() // 47
df_cleaned.filter($"cycle_type"==="RPT").select("battery_id").distinct().count() // 22
// there are some battteries with no RPT cycles

// number of cycles
df_cleaned.select("battery_id","cycle").distinct().count() // 7940, match 31760/4

//
val df_cycles = df_cleaned.select("battery_model","battery_id","cycle","cycle_type","charge_duration_sec","discharge_capacity_pct").distinct()

val df_soc_1 = df_cleaned.select("battery_model","battery_id","cycle","feature_1","feature_2","feature_3").filter($"soc_region" === 1).withColumnRenamed("feature_1","soc1_feature_1").withColumnRenamed("feature_2","soc1_feature_2").withColumnRenamed("feature_3","soc1_feature_3")

val df_soc_2 = df_cleaned.select("battery_model","battery_id","cycle","feature_1","feature_2","feature_3").filter($"soc_region" === 2).withColumnRenamed("feature_1","soc2_feature_1").withColumnRenamed("feature_2","soc2_feature_2").withColumnRenamed("feature_3","soc2_feature_3")

val df_soc_3 = df_cleaned.select("battery_model","battery_id","cycle","feature_1","feature_2","feature_3").filter($"soc_region" === 3).withColumnRenamed("feature_1","soc3_feature_1").withColumnRenamed("feature_2","soc3_feature_2").withColumnRenamed("feature_3","soc3_feature_3")

val df_soc_4 = df_cleaned.select("battery_model","battery_id","cycle","feature_1","feature_2","feature_3").filter($"soc_region" === 4).withColumnRenamed("feature_1","soc4_feature_1").withColumnRenamed("feature_2","soc4_feature_2").withColumnRenamed("feature_3","soc4_feature_3")

val df_combined_cycles = df_cycles.join(df_soc_1, Seq("battery_model","battery_id", "cycle")).join(df_soc_2, Seq("battery_model","battery_id", "cycle")).join(df_soc_3, Seq("battery_model","battery_id", "cycle")).join(df_soc_4, Seq("battery_model","battery_id", "cycle"))

val df_output = df_combined_cycles.orderBy($"battery_model",$"battery_id",$"cycle")
```

```scala
df_output.coalesce(1).write.saveAsTable("gbatteries_team_8.cleaned_data")
df_output.coalesce(1).write.option("header","true").csv("cleaned_data.csv")


// there are some data weird
// step 1: check if a battery has enough data
val df_battery_cycles_count = df_combined_cycles.groupBy("battery_id").agg(count(lit(1)) as "battery_count", count(when($"cycle_type"==="RPT",1)) as "RPT_count", count(when($"cycle_type"==="Regular",1)) as "Regular_count").orderBy("battery_id")

df_battery_cycles_count.show()

df_battery_cycles_count.coalesce(1).write.saveAsTable("gbatteries_team_8.battery_cycles_count")
df_battery_cycles_count.coalesce(1).write.option("header","true").csv("battery_cycles_count.csv")

// step 2: separate the data set in terms of cycle type to observe trend
val df_output_RPT = df_combined_cycles.filter($"cycle_type" === "RPT").select("battery_model","battery_id","cycle","discharge_capacity_pct").orderBy($"battery_model",$"battery_id",$"cycle")
val df_output_Regular = df_combined_cycles.filter($"cycle_type" === "Regular").select("battery_model","battery_id","cycle","discharge_capacity_pct").orderBy($"battery_model",$"battery_id",$"cycle")

df_output_RPT.coalesce(1).write.saveAsTable("gbatteries_team_8.cleaned_data_rpt")
df_output_RPT.coalesce(1).write.option("header","true").csv("cleaned_data_RPT.csv")

df_output_Regular.coalesce(1).write.saveAsTable("gbatteries_team_8.cleaned_data_regular")
df_output_Regular.coalesce(1).write.option("header","true").csv("cleaned_data_Regular.csv")

val rpt_raw = spark.table("gbatteries_team_8.cleaned_data_rpt").select("battery_id","cycle","discharge_capacity_pct")

// select only batteries having RPT cycles
val reg_raw = spark.table("gbatteries_team_8.cleaned_data_regular").select("battery_id","cycle").filter($"battery_id".isin("A8C2C39","A65D895","A2C0E4A","AFFE915","A947A53","A1D35B0","A74A962","AFD5A8F","AEBE15E","A716092","A18242C","A4E5D13","AEE8EF6","A231712","A6AD931","A35D29D","AC3C95D","A83B987","A73CF24","A0047B2","AD5B491","A43046B"))

// take last RPT cycle of consecutive RPT cycles as measurement (and label whether it is from a consecutive set)
// create column of next RPT cycle number and column of last RPT cycle number
val battery_id_window = Window.partitionBy($"battery_id").orderBy("cycle")
val lead_cycle = lead("cycle",1,0).over(battery_id_window)
var rpt_temp = rpt_raw.select($"*",lead_cycle as "next_RPT_cycle")
val last_cycle = lag("cycle",1,0).over(battery_id_window)
rpt_temp = rpt_temp.select($"*",last_cycle as "last_RPT_cycle")
// only keep records if next cycle is not RPT, and label if its last cycle is RPT
rpt_temp = rpt_temp.filter($"next_RPT_cycle" - $"cycle" =!= 1)
rpt_temp = rpt_temp.withColumn("consecutive_rpt",when(($"cycle" - $"last_RPT_cycle" === 1) && ($"cycle" =!= 1), 1).otherwise(0))

// eliminate regular data after reaching last RPT cycle
val rpt_last_cycle = rpt_temp.groupBy("battery_id").agg(max("cycle") as "last_rpt_cycle")
var reg_temp = reg_raw.join(rpt_last_cycle,"battery_id")
reg_temp = reg_temp.filter($"cycle" < $"last_rpt_cycle")

// exporting data
val rpt_final = rpt_temp.select("battery_id", "cycle", "discharge_capacity_pct", "consecutive_rpt").orderBy("battery_id","cycle")
val reg_final = reg_temp.select("battery_id", "cycle").orderBy("battery_id","cycle")

// rpt actual data for computing smape
val rpt_eol_3 = rpt_final.select("battery_id", "cycle", "discharge_capacity_pct")

rpt_eol_3.coalesce(1).write.saveAsTable("gbatteries_team_8.rpt_actual_v8")
rpt_eol_3.coalesce(1).write.option("header","true").csv("rpt_actual_v8.csv")


val df_rpt_raw_data = spark.table("gbatteries_team_8.cleaned_data_rpt")
val df_rpt_selected_data = df_rpt_raw_data.filter($"battery_id".isin("AEBE15E","AFD5A8F","A74A962","A1D35B0"
```

```
,"A947A53","AFFE915","A2C0E4A","A65D895","A8C2C39"))

// verify data counts
val check_1 = df_rpt_selected_data.groupBy("battery_id").agg(count(lit(1)) as "RPT_count").orderBy("battery_id")

// create column of next RPT cycle number
val lead_window = Window.partitionBy($"battery_id").orderBy("cycle")
val lead_cycle = lead("cycle",1,0).over(lead_window)
val df1 = df_rpt_selected_data.select($"*",lead_cycle as "next_RPT_cycle")
df1.select("cycle","next_RPT_cycle").show(25)

// next_RPT_cycle is 0 for final row, which is different from the null in Pyspark
// remove consecutive RPT cycles
val df2 = df1.filter($"next_RPT_cycle" - $"cycle" =!= 1).drop("next_RPT_cycle")
df2.show(25)

// split into training and testing set
val df_output = df2.select("battery_id","cycle","discharge_capacity_pct")
val df_output_training = df2.filter($"battery_id".isin("AEBE15E","AFD5A8F","A74A962","A1D35B0","A947A53","A2C0E4A","A65D895")).filter($"cycle" < 120)
val df_output_testing = df2.filter($"battery_id".isin("A8C2C39","AFFE915"))

// export data sets
df_output_training.coalesce(1).write.saveAsTable("gbatteries_team_8.lin_reg_training")
df_output_training.coalesce(1).write.option("header","true").csv("lin_reg_training.csv")

df_output_testing.coalesce(1).write.saveAsTable("gbatteries_team_8.lin_reg_testing")
df_output_testing.coalesce(1).write.option("header","true").csv("lin_reg_testing.csv")


// This part is tuning for threshold value for step 2 (deviation from line of training points) of enhanced linear regression
val df_training = spark.table("gbatteries_team_8.lin_reg_training")

// perform linear regression to each set of early RPT points to see how much the points deviate from the line generated

// boundary 0.05 eliminate a minor outlier but 0.043 would make the slope +
val df1 = df_training.filter($"battery_id"==="AEBE15E").select("cycle","discharge_capacity_pct")

// boundary 0.04 cycle65 for df2 would make perfect line
val df2 = df_training.filter($"battery_id"==="AFD5A8F").select("cycle","discharge_capacity_pct").filter($"cycle" =!= 65)

val assembler = new VectorAssembler().setInputCols(Array("cycle")).setOutputCol("features")

val df1_input = assembler.transform(df1).drop("cycle").withColumnRenamed("discharge_capacity_pct","label")
val lr = new LinearRegression()
val model1 = lr.fit(df1_input)
val df1_output = model1.transform(df1_input).withColumn("label - prediction",$"label" - $"prediction")
df1_output.show()

val df2_input = assembler.transform(df2).drop("cycle").withColumnRenamed("discharge_capacity_pct","label")
val lr = new LinearRegression()
val model2 = lr.fit(df2_input)
val df2_output = model2.transform(df2_input).withColumn("label - prediction",$"label" - $"prediction")
df2_output.show()
```