MACHINE LEARNING BASED GLUCOSE LEVEL PREDICTION USING CONTINUOUS GLUCOSE MONITORING

CREATIVE AND INNOVATIVE PROJECT REPORT

Submitted By

ADITYA RAMACHANDRAN (2019103502)

SHREYA ANANTH (2019103580)

CHIRAN JEEVI M P (2019103013)

in partial fulfilment of the requirements for the award of the

degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

COLLEGE OF ENGINEERING, GUINDY

ANNA UNIVERSITY: CHENNAI 600 025

JUNE 2022

ACKNOWLEDGEMENT

Foremost, we would like to express our sincere gratitude to our project guide, **Dr. Lalitha Devi K**, Teaching Fellow, Department of Computer Science and Engineering, College of Engineering Guindy, Chennai for her constant source of inspiration. We thank her for the continuous support and guidance which was instrumental in taking the project to successful completion.

We are grateful to **Dr. S. Valli**, Professor and Head, Department of Computer Science and Engineering, College of Engineering Guindy, Chennai for her support and for providing necessary facilities to carry out for our project.

We would also like to thank our friends and family for their encouragement and continued support. We would also like to thank the Almighty for giving us the moral strength to accomplish our task.



Aditya Ramachandran Shreya Ananth Chiran Jeevi MP

TABLE OF CONTENTS

ABSTRACT

Type-1 diabetes is a very common disorder which currently affects a variety of the world's population, majority being 0-16years of age. It's an auto-immune disorder in which the beta cells of the pancreas does not produce insulin and have to be given externally with the help of insulin shots. This project aims in predicting blood glucose levels that help in optimising closedloop insulin delivery systems.

It is done using machine learning algorithms that take as input, data from continuous glucose monitoring (CGM) and also the type and the time of meal along with its carbohydrate content. Depending on the above-mentioned features, an estimate of the blood glucose levels of the person at 15- and 60-minute intervals can be predicted. This would help the T1D person to adjust their bolus (fast-acting) and basal (longacting) dosage according to the time and type of their meal

1. INTRODUCTION

This project aims in predicting blood glucose levels that help in optimising closed-loop insulin delivery systems. The glucose level prediction is done first as a univariate time-series prediction using Auto Regressive Integrated Moving Average (ARIMA) model and Recurrent Neural Network (RNN) which takes as input, only the CGM data. Their performances are compared. Then, for more accurate predictions, several external factors like type of meal, insulin dosage and time of meal are also taken to predict glucose level as a multivariate time-series using Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). Depending on the above-mentioned features, an estimate of the blood glucose levels of the person at 15- and 60-minute intervals can be predicted. This would help the T1D person to adjust their bolus (fast-acting) and basal (long-acting) dosage according to the time and type of their meal.

1.1 OBJECTIVES

Our main objective is predicting blood glucose levels using machine learning algorithms that take as input, data from continuous glucose monitoring (CGM) and the type and the time of meal along with its carbohydrate content. Depending on the above-mentioned features, an estimate of the blood glucose levels of the person at 15- and 60-minute intervals can be predicted. Apart from this, our project further classifies the food into High GI & Low GI depending on how the food affects the variation of blood glucose level pattern.

1.2 PROBLEM STATEMENT

Type-1 diabetes is a very common disorder which currently affects a variety of the world's population, majority being 0-16years of age. It's an auto-immune disorder in which the beta cells of the pancreas do not produce insulin and have to be given externally with the help of insulin shots. To deliver the correct insulin dosage according to the food consumed, knowledge of the blood glucose level is required. Too much insulin would result in a Hypoglycemia and too little in a Hyperglycemia.

1.3 CHALLENGES IN THE SYSTEM

- The availability of a proper dataset is quite tough as different people have different eating habits (which consist of variety in food and the time at which they eat), and lifestyle (exercise, manual work, stationary work) which directly affects the body's blood glucose levels.
- So, in order to collect the dataset, a particular pattern has to be followed by the individuals involved so as to accurately study the glucose levels pattern without any other external factors affecting it.
- The readings of CGM (Continuous Glucose Monitoring) are also not 100% accurate

2. LITERATURE SURVEY

As mentioned earlier, the project aims in developing models to predict blood glucose levels in Type 1 Diabetes patients, at 15 minute and 60 minute intervals. We used various models to predict these values and also fed them with various inputs features. Many research papers have dealt with the topic of predicting blood glucose level using CGM.

The research paper "Artificial Neural Network for Blood Glucose Level Prediction" by Takoua HAMDI, Jaouher BEN ALI, Nader FNAIECH, has exploited Artificial Neural Network (ANN) for blood glucose level prediction of Type 1 Diabetes (T1D). In this paper, two objectives were targeted: First, defining automatically the optimal structure of the used ANN so define the optimal neurons in each layer. Second, adopting the obtained ANN for an accurate blood glucose prediction method. The main idea of the ANN was to use the N previous measures to predict the next measure. To validate the proposed method, real Continuous Glucose Monitoring (CGM) data of 12 patients were investigated. For the entire dataset, the average of the Root Mean Square Error (RMSE) (mg/dL) was found to be 6.43.

Deep learning has recently been applied in healthcare and medical research to achieve state-of-the-art results in a range of tasks including disease diagnosis, and patient state prediction among others.

"Using LSTMs to Learn Physiological Models of Blood Glucose Behaviour" by Sadegh Mirshekarian, Razvan Bunescu, Cindy Marling, and Frank Schwartz, is a research paper that approaches the same problem with a different learning model. In this paper, a recursive neural network (RNN) approach is described that uses long short- term memory (LSTM) units to learn a physiological model of blood glucose. The LSTM architecture was trained and evaluated on a dataset of 5 patients, containing approximately 400 days' worth of BG levels, insulin, and meal event data.

"Convolutional Recurrent Neural Networks for Glucose Prediction" by Kezhi Li, presents a deep learning model that is capable of forecasting glucose levels with leading accuracy for simulated patient cases (root-mean-square error (RMSE) = 9.38 ± 0.71 [mg/dL] over a 30-min horizon, RMSE = 18.87 ± 2.25 [mg/dL] over a 60-min horizon) and real patient cases (RMSE = 21.07 ± 2.35 [mg/dL] for 30 min, RMSE = 33.27 ± 4.79% for 60 min). The papers proposes a deep learning algorithm for glucose prediction using a multi-layer convolutional recurrent neural network (CRNN) architecture. The model is trained on data comprising CGM, carbohydrate and insulin data. After pre-processing, the time-aligned multi-dimensional time series data of BG, carbohydrate and insulin (other factors also can be considered) are fed to CRNN for training.

"Machine learning-based glucose prediction with use of continuous glucose and physical activity monitoring data: The Maastricht Study" by William P. T. M. van Doorn, used data from The Maastricht Study, an observational population-based cohort that comprises individuals with normal glucose metabolism, prediabetes, or type 2 diabetes. A random subset of individuals was used to train models in predicting glucose levels at 15- and 60-minute intervals based on either CGM data or both CGM and accelerometer data. In the remaining individuals, model performance was evaluated with root-mean-square error (RMSE), Spearman's correlation coefficient (rho) and surveillance error grid. A broad spectrum of hyperparameter combinations for this network was evaluated resulting in a multi-task LSTM architecture, consisting of three layers, including a dropout layer with a total of 56–104 neurons.

S.No.	Author-Publication Year	Methodology	Merits	Demerits
1	Takoua HAMDI - 2017	Artificial neural networks	the blood glucose is modeled automatically and adaptively by an ANN Separate models used for each patient	Doesn't consider other factors that can affect blood glucose level such as emotions, physical activity, meal intake and stress. Doesn't compare results with other models
2	Sadegh Mirshekarian - 2017	LSTM	Compares results with various models to account for the variability in the RNN results, the results were averaged over multiple runs to determine the impact of the patient history on the RNN performance, the LSTM approach was evaluated on the last 20 points in the dataset, varying the training history from 1 week, to 2 weeks, to 4 weeks,	Produces results with low accuracies. Doesn't consider performance on synthetic data set for agnostic performance

			to the entire patient history Considers meal intake and insulin dose effects in addition to cgm data	
3	Kezhi Li - 2020	CRNN	Compares the results of various models with different metrics other than RMSE.	Doesn't show performance of model when external factors apart from CGM are excluded.
4	William P. T. M. van Doorn - 2021	Multiple models were evaluated. Model of choice - LSTM	Compares the performance of various models Achieves high accuracy	Doesn't consider other factors that can affect blood glucose level such as emotions, physical activity, meal intake the main study population comprised individuals with NGM, prediabetes, or type 2 diabetes, who are generally not the target population for closed-loop insulin delivery systems.

3. PROPOSED APPROACH

Diabetes is a chronic illness characterised by the absence of glucose homeostasis. A healthy pancreas dynamically controls the release of insulin and glucagon hormones through the α-cells and β-cells respectively, in order to maintain euglycemia . In Type 1 diabetes, an autoimmune disease, the β-cells are compromised and therefore suffer from impaired production of insulin. This leads to periods of hyperglycaemia. Insulin therapy is needed to maintain BG levels in the advised target range.

The standard approach to diabetes management requires people actively taking BG measurements a handful of times throughout the day with a finger prick test - self monitoring of blood glucose. The recent development and uptake of continuous glucose monitoring (CGM) devices allow for improved sampling (5 minutes) of glucose measurements

In this project we utilise this CGM data and the various aspects that governs this blood glucose level to predict the BG in the next 15 minute and 60 minute from last observed data point. We have split the modules into two parts: one that uses the CGM data alone to predict the future values and another one that also takes into account external factors.

Before it is been given as input to the various models, the data has to be pre-processed to remove noises. The pre-processing is different for the different input models and datasets.

The Univariate module aims in working out the BG levels only using the CGM data. Two methods were chosen for predicting these values: SARIMAX and LSTM. These were first trained on synthetic data and then was trained on real world data from the OhioT1DM dataset.

The synthetic data was first pre-processed to remove null values, by replacing them with the mean of the other values. This was then passed as input to SARIMAX with the hyper parameters adjusted to this dataset. The same dataset was then given as input to the LSTM tuned similarly to suit the dataset.

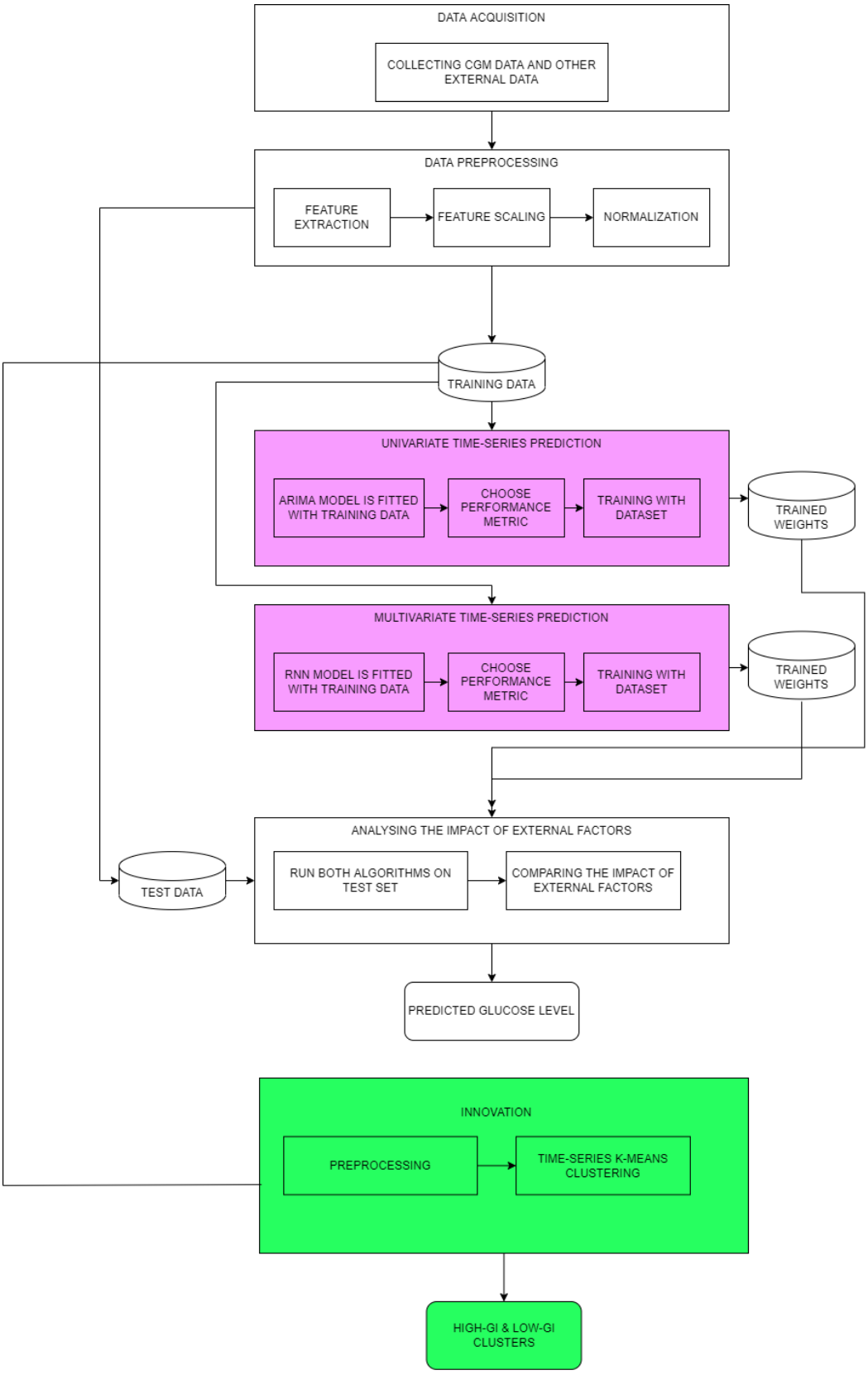
The OhioT1DM dataset was then pre-processed. The data of the training and test data of the sample patient was split into multiple .csv files corresponding to each day's data. The features that were selected to be included in these data frames were exercise data, bolus and basal insulin dose data, meal intake data and CGM data.

From the pre-processed OhioT1DM dataset for the patient 563, only the CGM data and time data was given as input to SARIMAX and LSTM model. The Multivariate module predicts the BG levels using meal intake and insulin dosage information in addition to CGM data. The data is further pre-processed into tensors in order to be given as input to the CRNN.

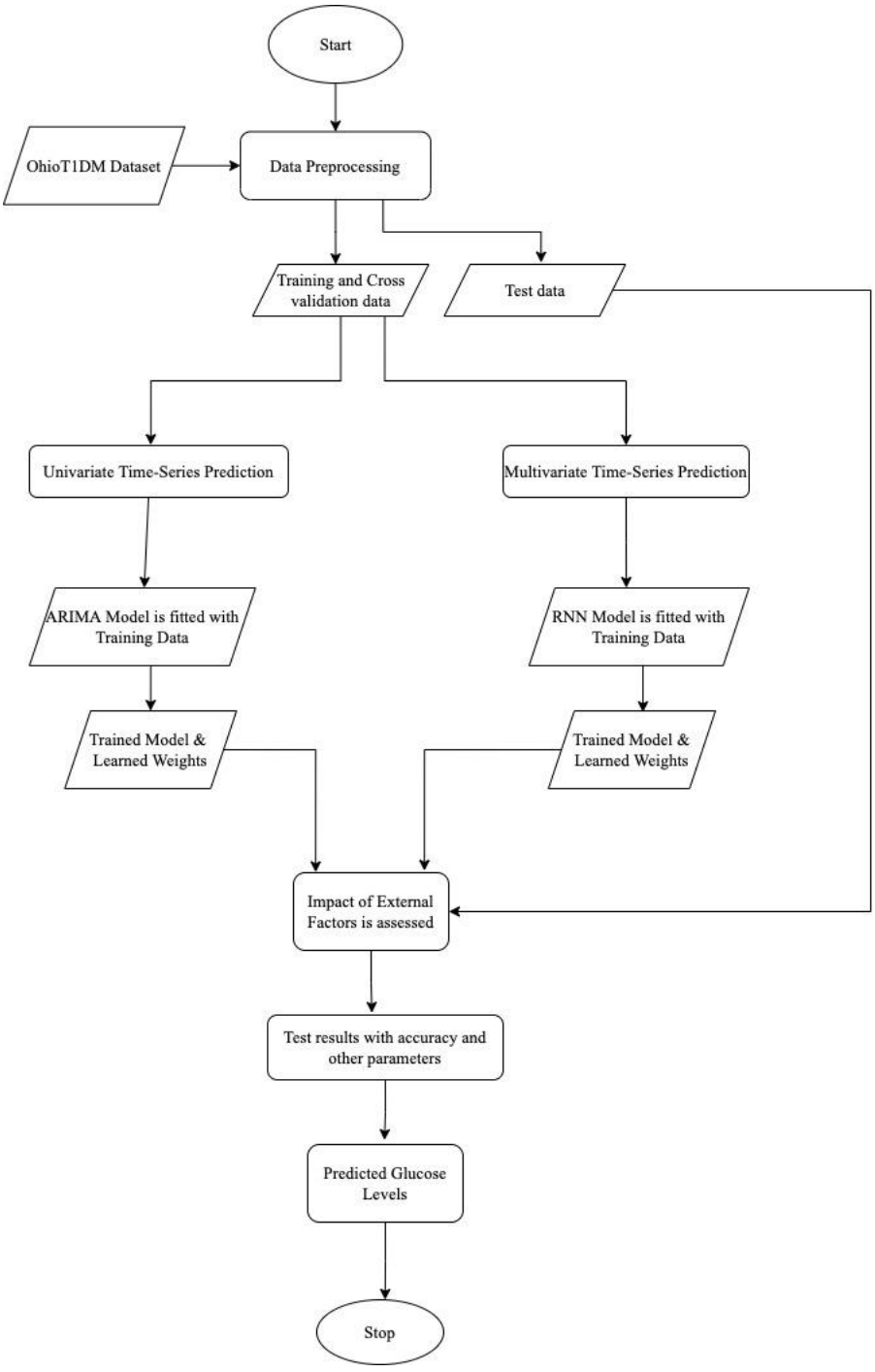
The models are evaluated on the entire date files in the test files. The metrics used for evaluation are average Root mean square error and average Mean square error. The evaluation is done on the sample patient's test data. Before these two values are displayed, a sample testing is done one of the test files (for one day) and the values at 15 minutes and 60 minutes for that day is highlighted and shown separately, along with the RMSE and MSE on that test data.

In addition to the 2 modules, we attempted to cluster the various meals taken into 2 groups and identify the low GI and high GI groups using graph analytics. The data was again pre-processed to extract the input feature vectors corresponding to each meal data. The CGM data from the time of meal intake till the time any external factors interfered were considered. This resulted in variable sized input vectors or time series data to be specific. The tslearn module was used to segregate this meal time series data into 2 clusters corresponding to low and high GI foods, using k means. The model was not evaluated due to its highly exploratory nature.

4. SYSTEM DESIGN



5. DETAILED MODULE DESIGN



Data is pre-processed by eliminating the null values and by substituting them with the mean of the rest of the values.

MODULES:

5.1 UNIVARIATE TIME-SERIES PREDICTION

This module predicts the blood glucose level by considering only CGM data. This is thus a Univariate time series prediction problem since it uses only the previous values of the time series to predict its future values.

Seasonality and trends are identified using seasonal_decompose and ad fuller tests respectively.

decompose_data = seasonal_decompose(data, model="additive", period =1,extrapolate_trend='freq')

result = adfuller(df.value)

It was identified that the data was a 'non-seasonal' time series. Since non-seasonal time series that exhibits patterns and is not a random white noise can be modelled with ARIMA (Auto Regressive Integrated Moving Average) it was chosen. ARIMA is a class of models that 'explains' a given time series based on its own past values, that is, its own lags and the lagged forecast errors, so that equation can be used to forecast future values.

model = SARIMAX(data.value, order=(p,q,d), seasonal_order=(p,q,d,0))

LSTM is also used to predict the value as the algorithm. LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series.

The accuracy produced by both the models are compared and presented as a plot.

Input: Pre-processed OHIO T1DM Data

Output: Final plot showing the difference in the values predicted by the 2 models and their accuracies.

5.2 MULTIVARIATE TIME-SERIES PREDICTION AND PERFORMANCE PREDICTION:

This module predicts the blood glucose level by also considering other external factors such as insulin dosage and meal intake data. This is to improve the accuracy of the models. Convolutional Neural Network(CNN) and Recurrent Neural Network(RNN) are used. The approach consists of pre-processing, feature extraction using CNN, time series prediction using LSTM and a signal converter.

Input: Pre-processed OHIO T1DM, Data

Output: Final plot showing the predicted value and the actual values, and the accuracy obtained.

5.3 CLASSIFYING FOOD AS HIGH GI OR LOW GI FOOD:

This is the innovative portion of the project. Based on the given meal intake data and the rate of increase observed in the blood glucose level, the food taken is classified as low GI or high GI. Based on the data available, this is modelled as an unsupervised learning problem.

Input: Pre-processed OHIO T1DM Data

Output: The meal intake is classified as either high-GI or low-GI food.

6. IMPLEMENTATION

6.1 SARIMAX MODEL FOR UNIVARIATE TIME-SERIES PREDICTION

The SARIMAX Model is for forecasting the future values of a particular curve based on past values of the given time-series. Four kinds of components help make a time series, and also they can affect our time series analysis if present in excess. So here, for this time series, we need to check more for the availability of components. The components are observed, trend, seasonal and residual values.

To perform forecasting using the ARIMA model, we required a stationary time series. Stationary time series is a time series that is unaffected by these four components. Most often, it happens when the data is non-stationary the predictions we get from the ARIMA model are worse or not that accurate.

Screenshots:

1. Imports

```
#training the model for one patient
from  statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.metrics import mean_squared_error
from pmdarima.arima import auto_arima
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


train = pd.read_csv('Awesome/Train1.csv')
test = pd.read_csv('Awesome/Test1.csv')
```

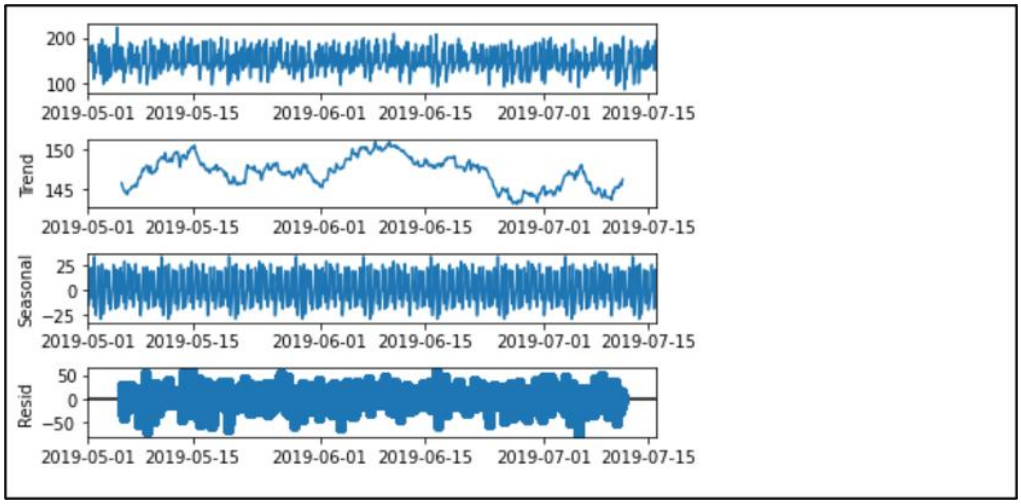
2. Cleaning data and pre-processing

```
#cleaning the data
#dropping Hour and Date column
train.drop(columns='Hour', inplace=True)
test.drop(columns='Hour', inplace=True)
train.drop(columns='Date', inplace=True)
test.drop(columns='Date', inplace=True)

#substituting the null values of glucose level with the mean of the other values, if present
n = train[train['BG'].isnull()].index.tolist()
if len(n)!=0:
    train['BG'][n] = np.mean(train['gl'])

n = test[test['BG'].isnull()].index.tolist()
if len(n)!=0:
    test['BG'][n] = np.mean(test['BG'])
```

3. Further pre-processing and seasonal decomposition

```
#changing the data type of the date_time column from object to datetime
train['Time'] = pd.to_datetime(train['Time'])
test['Time'] = pd.to_datetime(test['Time'])

#removing seconds to ensure periodicity
train['Time'] = pd.to_datetime(train['Time']).dt.floor('T')
test['Time'] = pd.to_datetime(test['Time']).dt.floor('T')

#creating new df for graphing purposes
train_plot = train.copy()
tdi = pd.DatetimeIndex(train_plot.Time)
tdi = pd.date_range(tdi[0],tdi[-1],freq='180S')
train_plot.set_index(tdi,inplace=True)
train_plot.drop(columns='Time',inplace=True)

decompose_data = seasonal_decompose(train_plot, period = 60*24*3, model="additive")
decompose_data.plot();
```

Output (For sample synthetic data):



4. Ad-fuller test and results

```
#checking for trend
result = adfuller(train.BG)
print(result)
```

```
(-20.255102149935205, 0.0, 13, 36466, {'1%': -3.4305293385723394, '5%': -2.8616192633175634, '10%': -2.566812189355308}, -405439.59667758516)
```

5. Fitting univariate data into SARIMAX model

```
model = SARIMAX(train['BG'],order=(5, 1, 1),seasonal_order=(1,1,1,7))
model_fit = model.fit()
```

6. Training, forecasting and finding errors (For sample synthetic data)

```
fc = model_fit.forecast(20, alpha=0.1)
fc = fc.reset_index()
fc.drop(columns='index',inplace=True)
testing = test.iloc[0:20,:]

mse = mean_squared_error(fc, testing['BG'])
print("Actual\tPredicted")
for i in range(20):
    print(test['BG'][i], fc['predicted_mean'][i])

print("\nValue after 15 min (actual, predicted): ", test['BG'][4], fc['predicted_mean'][4])
print("Value after 60 min (actual, predicted): ", test['BG'][19], fc['predicted_mean'][19])
print("Mean squared error: ", mse)
```

Output:

```
Actual  Predicted
150.42795 150.42767601053617
149.6819539 149.68058277633145
148.9578117 148.9536605151489
148.2554856 148.24581910320043
147.5749034 147.55573140109374
146.9159605 146.8818263517398
146.2785229 146.22236335112498
145.6624293 145.57577341138003
145.0674932 144.94078292958292
144.4935053 144.31643328738588
143.9402353 143.70205972316182
143.4074337 143.09725041408223
142.8948338 142.5018021445317
142.4021531 141.91567877659156
141.9290951 141.3387107262267
141.4753507 140.77056231780156
141.0405997 140.21066972839338
140.6245118 139.6583329507048
140.2267481 139.11268624369373
139.8469623 138.5726722689394

Value after 15 min (actual, predicted):  147.5749034 147.55573140109374
Value after 60 min (actual, predicted):  139.8469623 138.5726722689394
Mean squared error:  0.29681527451101974
```

Testing model on real world Ohio T1DM Dataset:

Prediction Code:

```
model = SARIMAX(training['cgm'], order=(3,1,4), seasonal_order=(3,1,2,7))
model_fit = model.fit()

fc60 = model_fit.forecast(12, alpha = 0.1).reset_index()
fc60.drop(columns='index', inplace=True)
mse60 = mean_squared_error(fc60, testing.iloc[0:12,1])

print('\n60 minute window:\nActual\tPredicted')
for i in range(12):
    print(testing['cgm'][trainlen+i], fc60['predicted_mean'][i])
```

Best Case Scenario: 1

```
data = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/Train/Date/date-2021-10-02.csv',
                   usecols=['time','cgm'])
```

```
print(data)
```

```
                  time  cgm
0    2021-10-02 12:29:00   73
1    2021-10-02 12:34:00   70
2    2021-10-02 12:39:00   68
3    2021-10-02 12:44:00   66
4    2021-10-02 12:49:00   65
..                   ...  ...
134  2021-10-02 23:39:00   80
135  2021-10-02 23:44:00   82
136  2021-10-02 23:49:00   85
137  2021-10-02 23:54:00   88
138  2021-10-02 23:59:00   89

[139 rows x 2 columns]
```

```
result = adfuller(data.cgm)
print(result)
print(len(data))
trainlen = int(0.9*len(data))
training = data[:trainlen]
testing = data[trainlen:]
```

```
(-1.9208396009968616, 0.3223218998014037, 4, 133, {'1%': -3.480500383888377, '5%': -2.8835279559405045, '10%': -2.57849571654700
7}, 626.7908336098786)
138
```

Running:

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =           13     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  2.55807D+00    |proj g|=  1.15992D-01

At iterate    5    f=  2.44857D+00    |proj g|=  5.87507D-02
```

Output:

```
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT

60 minute window:
Actual  Predicted
78 78.26217756618404
79 76.32439372962897
78 73.94563713539216
78 74.73136368796546
78 75.94745714357327
79 76.33812258219486
80 76.25335237152437
80 76.7159170518474
81 76.00453328905371
80 75.14601399983029
80 75.06883115795267
82 75.30851581655946
```

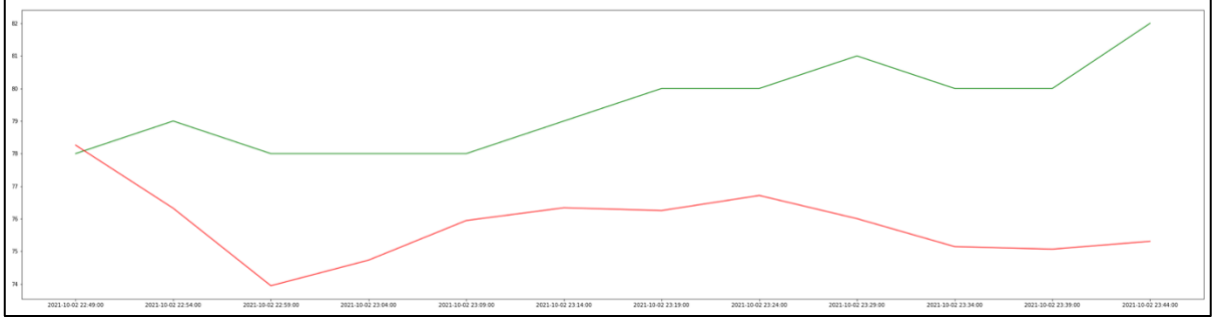
Error Metrics:

```
print("Best case performance: ")
print('Value after 15 min (Actual,predicted): ',testing['cgm'][trainlen+2], fc60['predicted_mean'][2])
print('Value after 60 min (Actual,predicted): ',testing['cgm'][trainlen+11], fc60['predicted_mean'][11])

print('Root mean squared error: ',math.sqrt(mse60))
print('Mean squared error: ',mse60)
```

```
Best case performance:
Value after 15 min (Actual,predicted):  78 73.94563713539216
Value after 60 min (Actual,predicted):  82 75.30851581655946
Root mean squared error:  3.958943407776755
Mean squared error:  15.673232905979026
```

```
fig, plt1= plt.subplots()
fig.set_figwidth(40)
fig.set_figheight(10)
plt1.plot(time[trainlen:trainlen+12], testing.iloc[0:12,1], color="green")
plt1.plot(time[trainlen:trainlen+12], fc60['predicted_mean'], color="red")
```

```
[<matplotlib.lines.Line2D at 0x151729270>]
```



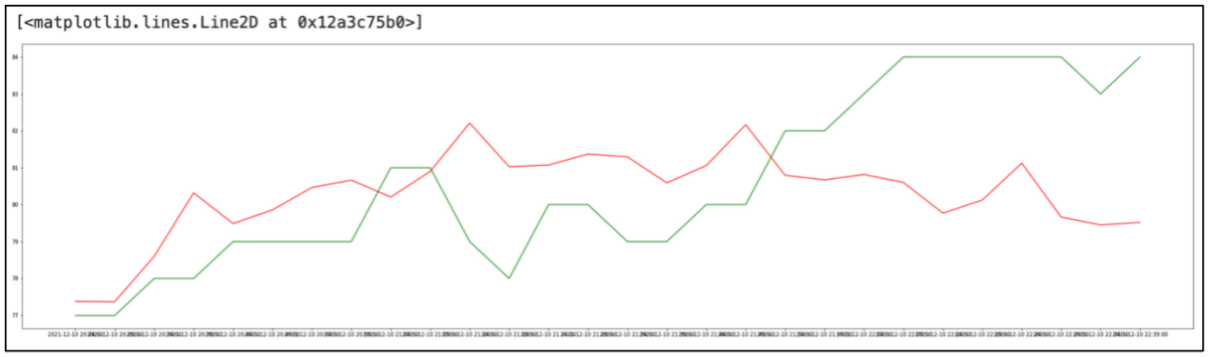
Green – Actual

Red – Predicted

Best Case Scenario: 2

```
import math
print('Value after 15 min (Actual,predicted): ',testing['cgm'][trainlen+2], fc['predicted_mean'][2])
print('Value after 60 min (Actual,predicted): ',testing['cgm'][trainlen+11], fc['predicted_mean'][11])
print('Root mean squared error: ',math.sqrt(mse))
```

```
Value after 15 min (Actual,predicted):  78 78.59903866463145
Value after 60 min (Actual,predicted):  78 81.02828565265654
Root mean squared error:  2.397020965539997
```



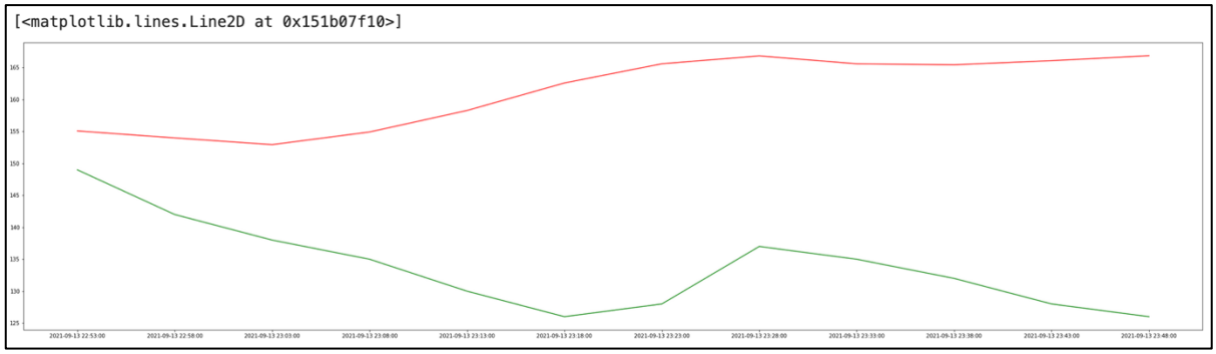
Green – Actual Red – Predicted

Worst Case Scenario:

```
data = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/Train/Date/date-2021-09-13.csv',
                   usecols=['time','cgm'])
```

```
print("Worst case performance: ")
print('Value after 15 min (Actual,predicted): ',testing['cgm'][trainlen+2], fc60['predicted_mean'][2])
print('Value after 60 min (Actual,predicted): ',testing['cgm'][trainlen+11], fc60['predicted_mean'][11])

print('Root mean squared error: ',math.sqrt(mse60))
print('Mean squared error: ',mse60)
```

```
Worst case performance:
Value after 15 min (Actual,predicted):  138 152.9446407980435
Value after 60 min (Actual,predicted):  126 166.86743623530495
Root mean squared error:  29.469883247802617
Mean squared error:  868.4740186391173
```



6.2 LSTM MODEL FOR UNIVARIATE TIME-SERIES PREDICTION

LSTM (Long Short-Term Memory) is a Recurrent Neural Network (RNN) based architecture that is widely used in natural language processing and time series forecasting. The LSTM rectifies a huge issue that recurrent neural networks suffer from: short-memory. Using a series of 'gates,' each with its own RNN, the LSTM manages to keep, forget or ignore data points based on a probabilistic model.

LSTMs also help solve exploding and vanishing gradient problems. While exploding and vanishing gradients are huge downsides of using traditional RNN's, LSTM architecture severely mitigates these issues. After a prediction is made, it is fed back into the model to predict the next value in the sequence.

Screenshots:

1. Imports

```
#training the model for one patient
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

train = pd.read_csv('Awesome/Train1.csv')
test = pd.read_csv('Awesome/Test1.csv')
```

2. Cleaning data and pre-processing

```
#cleaning the data
#dropping Hour and Date column
train.drop(columns='Hour', inplace=True)
test.drop(columns='Hour', inplace=True)
train.drop(columns='Date', inplace=True)
test.drop(columns='Date', inplace=True)

#substituting the null values of glucose level with the mean of the other values, if present
n = train[train['BG'].isnull()].index.tolist()
if len(n)!=0:
    train['BG'][n] = np.mean(train['gl'])

n = test[test['BG'].isnull()].index.tolist()
if len(n)!=0:
    test['BG'][n] = np.mean(test['BG'])
```

3. Converting dates in String format to datetime object

```
#changing the data type of the date_time column from object to datetime
train['Time'] = pd.to_datetime(train['Time'])
test['Time'] = pd.to_datetime(test['Time'])

#removing seconds to ensure periodicity
train['Time'] = pd.to_datetime(train['Time']).dt.floor('T')
test['Time'] = pd.to_datetime(test['Time']).dt.floor('T')
```

4. Importing LSTM algorithm and fitting model with dataset

```
#LSTM model to predict the glucose level
#look_back = 1

from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

p = train.iloc[:-1, 1]
q = train.iloc[1:,1]

model = Sequential()

model.add(LSTM(units=50, return_sequences=True, input_shape=(1, 1)))
model.add(Dropout(0.2))

model.add(LSTM(units=50, return_sequences=True))
model.add(Dropout(0.2))

model.add(LSTM(units=50, return_sequences=True))
model.add(Dropout(0.2))

model.add(LSTM(units=50))
model.add(Dropout(0.2))

model.add(Dense(units = 1))
model.compile(optimizer = 'adam', loss = 'mean_squared_error')
model.fit(p, q, epochs = 100, batch_size = 32)
```

5. Sample Epochs

```
Epoch 1/100
1140/1140 [==============================] - 5s 2ms/step - loss: 14297.2998
Epoch 2/100
1140/1140 [==============================] - 2s 2ms/step - loss: 6594.3374
Epoch 3/100
1140/1140 [==============================] - 2s 2ms/step - loss: 2708.2659
Epoch 4/100
1140/1140 [==============================] - 3s 2ms/step - loss: 1086.2450
Epoch 5/100
1140/1140 [==============================] - 3s 2ms/step - loss: 656.3010
Epoch 6/100
1140/1140 [==============================] - 2s 2ms/step - loss: 610.4348
Epoch 7/100
1140/1140 [==============================] - 2s 2ms/step - loss: 607.5351
Epoch 8/100
1140/1140 [==============================] - 2s 2ms/step - loss: 609.7151
Epoch 9/100
1140/1140 [==============================] - 2s 2ms/step - loss: 606.4655
Epoch 10/100
1140/1140 [==============================] - 2s 2ms/step - loss: 607.5217
Epoch 11/100
1140/1140 [==============================] - 2s 2ms/step - loss: 610.8523
Epoch 12/100
1140/1140 [==============================] - 2s 2ms/step - loss: 594.8352
Epoch 13/100
1140/1140 [==============================] - 2s 2ms/step - loss: 251.0123
Epoch 14/100
1140/1140 [==============================] - 2s 2ms/step - loss: 167.9023
Epoch 15/100
1140/1140 [==============================] - 2s 2ms/step - loss: 148.9254
```

```
Epoch 90/100
1140/1140 [==============================] - 80s 70ms/step - loss: 81.7818
Epoch 91/100
1140/1140 [==============================] - 2s 2ms/step - loss: 79.6879
Epoch 92/100
1140/1140 [==============================] - 2s 2ms/step - loss: 81.1619
Epoch 93/100
1140/1140 [==============================] - 2s 2ms/step - loss: 79.3720
Epoch 94/100
1140/1140 [==============================] - 2s 2ms/step - loss: 79.5907
Epoch 95/100
1140/1140 [==============================] - 2s 2ms/step - loss: 78.7351
Epoch 96/100
1140/1140 [==============================] - 390s 343ms/step - loss: 78.3844
Epoch 97/100
1140/1140 [==============================] - 2s 2ms/step - loss: 77.6731
Epoch 98/100
1140/1140 [==============================] - 2s 2ms/step - loss: 78.0048
Epoch 99/100
1140/1140 [==============================] - 2s 2ms/step - loss: 78.8297
Epoch 100/100
1140/1140 [==============================] - 2s 2ms/step - loss: 78.4035
```

6. Model Predictions

```
p = test.iloc[:-1, 1]
q = test.iloc[1:,1]

predictions = model.predict(p)
```

7. Printing sample predicted vs Actual data

```
pred = []
for i in range(len(predictions)):
    pred.append(predictions[i][0])
print("Actual Predicted")
for i in range(len(pred)):
    print(q[i+1], pred[i])
```

```
Actual Predicted
149.6819539 150.37317
148.9578117 149.62454
148.2554856 148.90364
147.5749034 148.2103
146.9159605 147.54402
146.2785229 146.90466
145.6624293 146.29196
145.0674932 145.70563
144.4935053 145.14502
143.9402353 144.60973
143.4074337 144.09904
142.8948338 143.61208
142.4021531 143.14774
141.9290951 142.70499
141.4753507 142.28282
141.0405997 141.8801
140.6245118 141.49565
140.2267481 141.12856
139.8469623 140.77779
139.4848016 140.44264
139.1399078 140.12228
138.8119182 139.81622
138.5004666 139.524
```

8. Finding Mean Squared Error

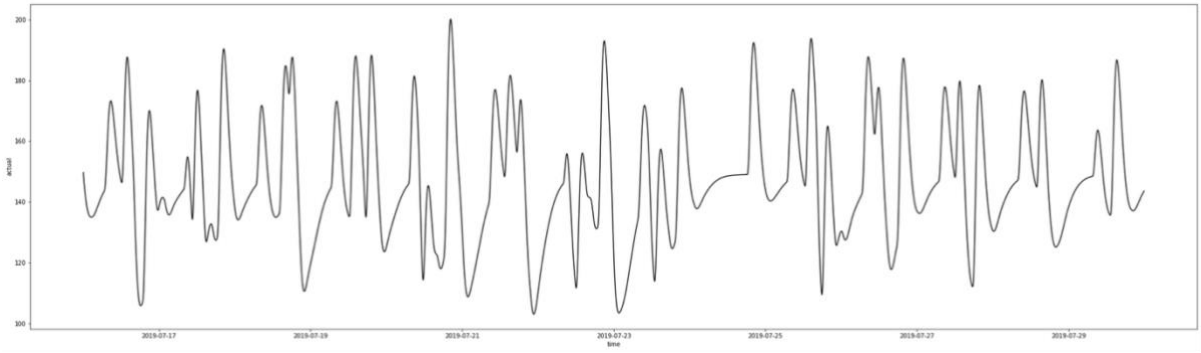
```
mse = mean_squared_error(pred, q)
print(mse)
```

```
1.0972947323919702
```

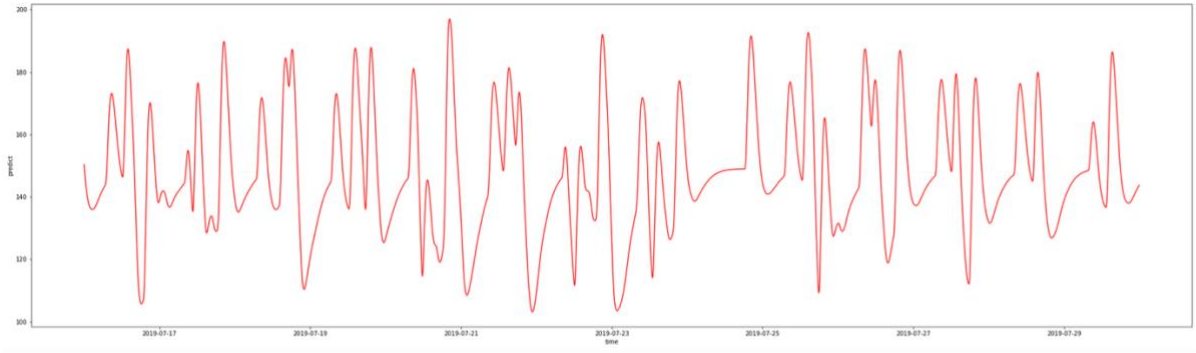
9. Printing Predicted(Forecasted) vs Actual Data

```
time = test.iloc[1:,0]
fig1, act = plt.subplots()
fig2, predict = plt.subplots()
fig1.set_figwidth(35)
fig1.set_figheight(10)
fig2.set_figwidth(35)
fig2.set_figheight(10)

act.plot(time, q, color="black")
act.set_xlabel("time")
act.set_ylabel("actual")
predict.plot(time, pred, color="red")
predict.set_xlabel("time")
predict.set_ylabel("predict")
```

Actual Data



Forecasted Data



Testing model on real world Ohio T1DM Dataset:

Training data:

```
import os
path = "/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/Train/Date"
df = []
for file in os.listdir(path):
    if file==".DS_Store":
        continue
    full_path = os.path.join(path,file)
    t = pd.read_csv(full_path, index_col=0)
    df.append(t)
```

Sample:

```
#changing the data type of the date_time column from object to datetime
for d in df:
    d['time'] = pd.to_datetime(d['time']).dt.floor('T').dt.time
```

```
print(df[0])
```

```
           date      time  cgm  bas  bol  meal_carb meal_type  ex
6744  2021-10-09  00:00:00  123    0  0.0          0      None   0
6745  2021-10-09  00:05:00  124    0  0.0          0      None   0
6746  2021-10-09  00:10:00  125    0  0.0          0      None   0
6747  2021-10-09  00:15:00  121    0  0.0          0      None   0
6748  2021-10-09  00:20:00  121    0  0.0          0      None   0
...          ...       ...  ...  ...  ...        ...       ...  ..
7027  2021-10-09  23:35:00  117    0  0.0          0      None   0
7028  2021-10-09  23:40:00  111    0  0.0          0      None   0
7029  2021-10-09  23:45:00  106    0  0.0          0      None   0
7030  2021-10-09  23:50:00  105    0  0.0          0      None   0
7031  2021-10-09  23:55:00  102    0  0.0          0      None   0

[288 rows x 8 columns]
```

Prediction:

```
#LSTM model to predict the glucose level
#look_back = 1

from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

p = data.iloc[:-1, 1]
q = data.iloc[1:,1]

model1 = Sequential()

model1.add(LSTM(units=50, return_sequences=True, input_shape=(1, 1)))
model1.add(Dropout(0.2))

model1.add(LSTM(units=50, return_sequences=True))
model1.add(Dropout(0.2))

model1.add(LSTM(units=50, return_sequences=True))
model1.add(Dropout(0.2))

model1.add(LSTM(units=50))
model1.add(Dropout(0.2))

model1.add(Dense(units = 1))
model1.compile(optimizer = 'adam', loss = 'mean_squared_error')

for i in range(350):
    print(f'Epoch - {i}')
    for d in df:
        p = d.iloc[:-1,1]
        q = d.iloc[1:, 1]
        model1.fit(p, q, epochs = 1, batch_size = 32)
```

Testing for particular data:

```
data1 = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/Test/Date/date-2021-10-22.csv',
                    usecols=['time','cgm'])
p = data1.iloc[:12, 1]
q = data1.iloc[1:13, 1]
time = data1.iloc[1:13, 0]
predictions = model1.predict(p)
```

```
pred = []
for i in range(len(predictions)):
    pred.append(predictions[i][0])
print("Actual Predicted")
for i in range(len(pred)):
    print(q[i+1], pred[i])
```

```
Actual Predicted
141 [141.35184]
136 [140.34929]
131 [135.3265]
125 [130.29327]
119 [124.253365]
114 [118.22638]
109 [113.21781]
106 [108.21806]
112 [105.2182]
114 [111.21733]
114 [113.21781]
110 [113.21781]
```

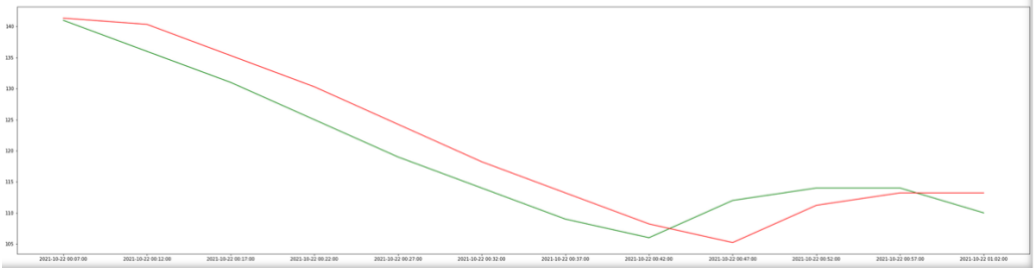
Error Metrics:

```
mse = mean_squared_error(pred, q)
print("\nValue after 15 min (actual, predicted): ", q[3], pred[2])
print("Value after 60 min (actual, predicted): ", q[12], pred[11])

print('\nRoot mean squared error: ',math.sqrt(mse))
print('Mean squared error: ',mse)
```

```
Value after 15 min (actual, predicted):  131 [135.3265]
Value after 60 min (actual, predicted):  110 [113.21781]

Root mean squared error:  4.068676700390685
Mean squared error:  16.554130092302028
```

```
fig, plt1= plt.subplots()
fig.set_figwidth(40)
fig.set_figheight(10)
plt1.plot(time, q, color="green")
plt1.plot(time, pred, color="red")
```

```
[<matplotlib.lines.Line2D at 0x290114760>]
```



Average Mean Squared errors over entire test data:

```
import os
path = "/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/Test/Date"
count = 0
mmse = 0
mrmse = 0
for file in os.listdir(path):
    if file==".DS_Store":
        continue
    count = count + 1
    full_path = os.path.join(path,file)
    data1 = pd.read_csv(full_path, index_col=0)
    p = data1.iloc[:11, 2]
    q = data1.iloc[1:12, 2]
    time = data1.iloc[1:12, 0]
    predictions = model1.predict(p)
    pred = []
    for i in range(len(predictions)):
        pred.append(predictions[i][0])
    mse = mean_squared_error(pred, q)
    mmse += mse
    mrmse += math.sqrt(mse)
```

```
mmse/=count
mrmse/=count

print("Performance for 60 min prediction window: ")
print("Average mean squared error: ", mmse)
print("Average root mean squared error: ", mrmse)
```

```
Performance for 60 min prediction window:
Average mean squared error:  10.628844986232782
Average root mean squared error:  2.881842135437121
```

6.3 PRE-PROCESSING MULTI-VARIATE DATA BY PARSING XML FILE USING XML.SAX

1. XML Data

```
<patient id="559" weight="99" insulin_type="Novalog">
  <glucose_level>
    ...
  </glucose_level>
  <finger_stick>
    ...
  </finger_stick>
  <basal>
    ...
  </basal>
  <temp_basal>
    ...
  </temp_basal>
  <bolus>
    ...
  </bolus>
  <meal>
    ...
  </meal>
  <sleep>
    ...
  </sleep>
  <work>
    ...
  </work>
  <stressors>
     <event ts="22-12-2021 17:03:00" type=" " description=" "/>
  </stressors>
  <hypo_event>
    ...
  </hypo_event>
  <illness>
    ...
  </illness>
  <exercise>
    ...
  </exercise>
  <basis_heart_rate>
    ...
  </basis_heart_rate>
  <basis_gsr>
    ...
  </basis_gsr>
  <basis_skin_temperature>
    ...
  </basis_skin_temperature>
  <basis_air_temperature>
    ...
  </basis_air_temperature>
  <basis_steps>
    ...
  </basis_steps>
  <basis_sleep>
    ...
  </basis_sleep>
</patient>
```

Sample fields

```
▼<patient id="559" weight="99" insulin_type="Novalog">
  ▼<glucose_level>
     <event ts="07-12-2021 01:17:00" value="101"/>
     <event ts="07-12-2021 01:22:00" value="98"/>
     <event ts="07-12-2021 01:27:00" value="104"/>
     <event ts="07-12-2021 01:32:00" value="112"/>
     <event ts="07-12-2021 01:37:00" value="120"/>
     <event ts="07-12-2021 01:42:00" value="127"/>
     <event ts="07-12-2021 01:47:00" value="135"/>
     <event ts="07-12-2021 01:52:00" value="142"/>
     <event ts="07-12-2021 01:57:00" value="140"/>
     <event ts="07-12-2021 02:02:00" value="145"/>
     <event ts="07-12-2021 02:07:00" value="148"/>
     <event ts="07-12-2021 02:12:00" value="151"/>
     <event ts="07-12-2021 02:17:00" value="150"/>
     <event ts="07-12-2021 02:22:00" value="124"/>
     <event ts="07-12-2021 02:27:00" value="130"/>
     <event ts="07-12-2021 02:32:00" value="127"/>
     <event ts="07-12-2021 02:37:00" value="121"/>
     <event ts="07-12-2021 02:42:00" value="115"/>
     <event ts="07-12-2021 02:47:00" value="111"/>
     <event ts="07-12-2021 02:52:00" value="109"/>
     <event ts="07-12-2021 02:57:00" value="103"/>
     <event ts="07-12-2021 03:02:00" value="89"/>
     <event ts="07-12-2021 03:07:00" value="76"/>
```

```
▼<basal>
   <event ts="07-12-2021 00:00:00" value="0.65"/>
   <event ts="07-12-2021 04:00:00" value="0.73"/>
   <event ts="07-12-2021 08:00:00" value="1.15"/>
   <event ts="07-12-2021 11:00:00" value="0.9"/>
   <event ts="08-12-2021 00:00:00" value="0.65"/>
   <event ts="08-12-2021 04:00:00" value="0.73"/>
   <event ts="08-12-2021 08:00:00" value="1.15"/>
   <event ts="08-12-2021 11:00:00" value="0.9"/>
   <event ts="08-12-2021 18:00:00" value="1.25"/>
   <event ts="11-12-2021 00:00:00" value="0.65"/>
   <event ts="11-12-2021 04:00:00" value="0.73"/>
   <event ts="11-12-2021 08:00:00" value="1.15"/>
   <event ts="11-12-2021 11:00:00" value="0.9"/>
   <event ts="12-12-2021 00:00:00" value="0.65"/>
   <event ts="12-12-2021 04:00:00" value="0.73"/>
   <event ts="12-12-2021 08:00:00" value="1.15"/>
   <event ts="12-12-2021 11:00:00" value="0.9"/>
   <event ts="12-12-2021 18:00:00" value="1.25"/>
   <event ts="13-12-2021 00:00:00" value="0.65"/>
   <event ts="13-12-2021 04:00:00" value="0.73"/>
   <event ts="13-12-2021 08:00:00" value="1.15"/>
   <event ts="13-12-2021 11:00:00" value="0.9"/>
   <event ts="13-12-2021 18:00:00" value="1.25"/>
   <event ts="16-12-2021 00:00:00" value="0.65"/>
   <event ts="16-12-2021 04:00:00" value="0.73"/>
```

2. Imports

```
import pandas as pd
import numpy as np
import xml.sax
```

3. Creating CGMHandler class to parse xml file

```
#creating ContentHandler class to parse xml file
class CGMHandler(xml.sax.ContentHandler):
    label= ""
    gl = { "time" : [] , "value" : []}
    bas = { "time" : [] , "value" :[]}
    bol = { "time" : [], "doze" : [], "carb" : []}
    meal = { "type" : [], "time" : [], "carb" : []}
    bs = { "beg" : [], "end" : [], "q" : []}
    slp = { "beg" : [], "end" : [], "q" : []}
    wrk = { "beg" : [], "end" : [], "int" : []}
    ex  = { "time" : [], "dur" : [], "int" : []}

    def __init__(self):
        self.CurrentData = ""

    def startElement(self, tag, attributes):
        if tag != "event":
            self.label = tag
        self.CurrentData = tag
        if tag == "event":
            if self.label == "glucose_level":
                ts = attributes["ts"]
                val =  attributes["value"]
                self.gl["time"].append(ts)
                self.gl["value"].append(val)


            elif self.label == "basal":
                ts = attributes["ts"]
                val =  attributes["value"]
                self.bas["time"].append(ts)
                self.bas["value"].append(val)
```

```
            elif self.label == "bolus":
                ts = attributes["ts_begin"]
                doze =  attributes["dose"]
                car = attributes["bwz_carb_input"]
                self.bol["time"].append(ts)
                self.bol["doze"].append(doze)
                self.bol["carb"].append(car)

            elif self.label == "meal":
                ts = attributes["ts"]
                typ =  attributes["type"]
                car = attributes["carbs"]
                self.meal["time"].append(ts)
                self.meal["type"].append(typ)
                self.meal["carb"].append(car)

            elif self.label == "sleep":
                b = attributes["ts_begin"]
                e =  attributes["ts_end"]
                q = attributes["quality"]
                self.slp["beg"].append(b)
                self.slp["end"].append(e)
                self.slp["q"].append(q)

            elif self.label == "work":
                b = attributes["ts_begin"]
                e =  attributes["ts_end"]
                i = attributes["intensity"]
                self.wrk["beg"].append(b)
                self.wrk["end"].append(e)
                self.wrk["int"].append(i)
```

```
            elif self.label == "exercise":
                ts = attributes["ts"]
                d =  attributes["duration"]
                i = attributes["intensity"]
                self.ex["time"].append(ts)
                self.ex["dur"].append(d)
                self.ex["int"].append(i)

            elif self.label == "basis_sleep":
                b = attributes["tbegin"]
                e =  attributes["tend"]
                q = attributes["quality"]
                self.bs["beg"].append(b)
                self.bs["end"].append(e)
                self.bs["q"].append(q)
```

4. Parsing the input XML file for one patient's record

```
#parsing the input xml file for one patient's record
parser = xml.sax.make_parser()
parser.setFeature(xml.sax.handler.feature_namespaces, 0)
Handler = CGMHandler()
parser.setContentHandler(Handler)
parser.parse("/Users/shreyaananth/Desktop/College/CIP/Code/OhioT1DM/2018/train/559-ws-training.xml")
```

5. Creating a data frame for each type of data given in xml file

```
#creating a dataframe for each type of data given in xml file
gl = pd.DataFrame(Handler.gl)
bas = pd.DataFrame(Handler.bas)
bol = pd.DataFrame(Handler.bol)
meal = pd.DataFrame(Handler.meal)
bs = pd.DataFrame(Handler.bs)
slp = pd.DataFrame(Handler.slp)
wrk = pd.DataFrame(Handler.wrk)
ex = pd.DataFrame(Handler.ex)
```

6. Saving data frames to csv files to modify certain aspects manually

```
#saving the dataframe into csv files to modify certain aspects manually
gl.to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/gl.csv")
bas.to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/bas.csv")
bol.to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/bol.csv")
meal.to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/meal.csv")
bs.to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/bs.csv")
slp.to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/slp.csv")
wrk.to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/wrk.csv")
ex.to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/ex.csv")

#changing the format of all dates into %yyyy-%mm-%dd manually
```

7. Changing time in String format to datetime object

```
gl = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/gl.csv',index_col=0)
bas = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/bas.csv',index_col=0)
bol = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/bol.csv',index_col=0)
meal = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/meal.csv',index_col=0)

#processing the time column: changing from string to datetime object
gl['time'] = pd.to_datetime(gl['time'],infer_datetime_format=True)
bas['time'] = pd.to_datetime(bas['time'],infer_datetime_format=True)
bol['time'] = pd.to_datetime(bol['time'],infer_datetime_format=True)
meal['time'] = pd.to_datetime(meal['time'],infer_datetime_format=True)
```

8. Updated Glucose level dataframe

```
#data after manual correction
gl
```

	time	value
0	2021-12-07 01:17:00	101
1	2021-12-07 01:22:00	98
2	2021-12-07 01:27:00	104
3	2021-12-07 01:32:00	112
4	2021-12-07 01:37:00	120
...	...	...
10791	2022-01-17 23:36:00	161
10792	2022-01-17 23:41:00	164
10793	2022-01-17 23:46:00	168
10794	2022-01-17 23:51:00	172
10795	2022-01-17 23:56:00	176

10796 rows × 2 columns

9. Extracting date from datetime object

```
#extracting date from time column to facilitate grouping of data according to date
gl['date'] = gl['time'].dt.date
bas['date'] = bas['time'].dt.date
bol['date'] = bol['time'].dt.date
meal['date'] = meal['time'].dt.date
```

10. Creating a new dataframe combining all the data into one table

```
#creating a new dataframe that acts as a template for combining all the patient data into one table
data = pd.DataFrame()
data['date'] = gl['date']
data['time'] = gl['time']
data['cgm'] = gl['value']
data['bas'] = [0]*len(data)
data['bol'] = [0]*len(data)
data['meal_carb'] = [0]*len(data)
data['meal_type'] = ["None"]*len(data)
```

11. Creating dataframe for each day's data

```
#creating dataframes for each day's data from the template dataframe
grouped_df = data.groupby(data.date)
keys = grouped_df.indices.keys()
keys = list(keys)

date_df = []
for i in range(len(keys)):
    df = grouped_df.get_group(keys[i])
    date_df.append(df)
```

12. Populating the dataframe with the CGM time as the reference

```
#populating the dataframes with data
for i in range(len(bas)):
    for k in range(len(date_df)):
        f = 0
        for j in range(len(date_df[k])-1):
            if bas.iloc[i,0]> date_df[k].iloc[j,1] and bas.iloc[i,0] <= date_df[k].iloc[j+1,1]:
                date_df[k].iloc[j+1,3] = bas.iloc[i,1]
                f = 1
                break
        if f==1:
            break

for i in range(len(bol)):
    for k in range(len(date_df)):
        f = 0
        if f==1:
            break
        for j in range(len(date_df[k])-1):
            if bol.iloc[i,0]> date_df[k].iloc[j,1] and bol.iloc[i,0] <= date_df[k].iloc[j+1,1]:
                date_df[k].iloc[j+1,4] = bol.iloc[i,1]
                f = 1
                break
        if f==1:
            break

for i in range(len(meal)):
    for k in range(len(date_df)):
        f = 0
        if f==1:
            break
        for j in range(len(date_df[k])-1):
            if meal.iloc[i,1]> date_df[k].iloc[j,1] and meal.iloc[i,1] <= date_df[k].iloc[j+1,1]:
                date_df[k].iloc[j+1,5] = meal.iloc[i,2]
                date_df[k].iloc[j+1,6] = meal.iloc[i,0]
                f = 1
                break
        if f==1:
            break
```

13. Creating csv file for each day's record

```
#creating csv file for each day's record
for i in range(len(keys)):
    string = "date-" + str(keys[i])
    date_df[i].to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/Date/"+string+".csv")

#sample dataframe from the created csv file
string = "/Users/shreyaananth/Desktop/College/CIP/Code/Data/Date/date-" + str(keys[0]) + ".csv"
df = pd.read_csv(string, index_col=0)
df
```

Sample dataframe

	date	time	cgm	bas	bol	meal_carb	meal_type
0	2021-12-07	2021-12-07 01:17:00	101	0.0	0.0	0	None
1	2021-12-07	2021-12-07 01:22:00	98	0.0	0.0	0	None
2	2021-12-07	2021-12-07 01:27:00	104	0.0	0.0	0	None
3	2021-12-07	2021-12-07 01:32:00	112	0.0	0.0	0	None
4	2021-12-07	2021-12-07 01:37:00	120	0.0	0.0	0	None
...	...	...	...	...	...	...	...
223	2021-12-07	2021-12-07 19:52:00	96	0.0	0.0	0	None
224	2021-12-07	2021-12-07 19:57:00	93	0.0	0.0	0	None
225	2021-12-07	2021-12-07 20:02:00	86	0.0	0.0	0	None
226	2021-12-07	2021-12-07 20:07:00	86	0.0	0.0	0	None
227	2021-12-07	2021-12-07 20:12:00	86	0.0	0.0	0	None

228 rows × 7 columns

6.4 MULTI-VARIATE TIME SERIES MODEL TRAINING WITH CRNN

1. Imports

```
#Imports
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers import TimeDistributed
from keras.layers import Dropout
from keras.layers import LSTM
```

2. Pre-processing the dataframe

```
#Preprocessing the dataframe for training after extracting it from csv file
datasets = []
cgm_data = []
import os
filepath = ["date-2021-12-07.csv","date-2021-12-08.csv","date-2021-12-09.csv","date-2021-12-10.csv"]
path = "/Users/shreyaananth/Desktop/College/CIP/Code/Data/Date"
for file in os.listdir(path):
    if file in filepath:
        full_path = os.path.join(path,file)
        df = pd.read_csv(full_path, index_col=0)
        df.drop(columns='date',inplace=True)
        df.drop(columns='time',inplace=True)
        df.drop(columns='meal_type',inplace=True)
        cgm = np.array(df['cgm'])
        bas = np.array(df['bas'])
        bol = np.array(df['bol'])
        meal_carb = np.array(df['meal_carb'])
        cgm = cgm.reshape((len(cgm),1))
        bas = bas.reshape((len(bas), 1))
        bol = bol.reshape((len(bol), 1))
        meal_carb = meal_carb.reshape((len(meal_carb), 1))
        dataset = np.hstack((bas,bol,meal_carb,cgm))
        cgm = cgm[1:]
        dataset = dataset[0:-1]
        shp = np.shape(dataset)
        dataset = dataset.reshape((shp[0], 1, 4, 1))
        datasets.append(dataset)
        cgm_data.append(cgm)
```

3. Creating CRNN Model

```
#Creating CRNN Model
model = Sequential()
model.add(Conv2D(8, kernel_size=(12,12), padding='same',activation="relu", input_shape = (1,4,1)))
model.add(MaxPooling2D(1))
model.add(Conv2D(7, kernel_size=(16,6), padding='same',activation="relu"))
model.add(MaxPooling2D(1))
model.add(Conv2D(5, kernel_size=(8,8), padding='same',activation="relu"))
model.add(MaxPooling2D(1))
model.add(Dropout(0.2))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(68, return_sequences=True))
model.add(Dropout(0.2))
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

4. Training dataset

```
#Training the dataset for 929 epochs
for j in range(929):
    print("Epoch ",j)
    for i in range(len(datasets)):
        print("Training model on ", filepath[i])
        model.fit(datasets[i], cgm_data[i], epochs=1)
```

5. Sample epochs

```
Epoch  0
Training model on  date-2021-12-07.csv
2022-05-18 12:46:14.767323: W tensorflow/core/platform/profile_utils/cpu_utils.cc:128] Failed to get CPU frequency:
0 Hz
9/9 [==============================] - 1s 2ms/step - loss: 32763.9609
Training model on  date-2021-12-08.csv
9/9 [==============================] - 0s 2ms/step - loss: 37965.4023
Training model on  date-2021-12-09.csv
9/9 [==============================] - 0s 2ms/step - loss: 24647.9238
Training model on  date-2021-12-10.csv
8/8 [==============================] - 0s 2ms/step - loss: 13405.9463
Epoch  1
Training model on  date-2021-12-07.csv
9/9 [==============================] - 0s 2ms/step - loss: 29831.0664
Training model on  date-2021-12-08.csv
9/9 [==============================] - 0s 2ms/step - loss: 32941.6133
Training model on  date-2021-12-09.csv
9/9 [==============================] - 0s 2ms/step - loss: 19310.8125
```

```
Epoch  927
Training model on  date-2021-12-07.csv
9/9 [==============================] - 0s 2ms/step - loss: 92.5902
Training model on  date-2021-12-08.csv
9/9 [==============================] - 0s 2ms/step - loss: 106.4879
Training model on  date-2021-12-09.csv
9/9 [==============================] - 0s 2ms/step - loss: 70.0145
Training model on  date-2021-12-10.csv
8/8 [==============================] - 0s 2ms/step - loss: 103.4788
Epoch  928
Training model on  date-2021-12-07.csv
9/9 [==============================] - 0s 2ms/step - loss: 96.2685
Training model on  date-2021-12-08.csv
9/9 [==============================] - 0s 2ms/step - loss: 140.7776
Training model on  date-2021-12-09.csv
9/9 [==============================] - 0s 2ms/step - loss: 94.0831
Training model on  date-2021-12-10.csv
8/8 [==============================] - 0s 2ms/step - loss: 114.7952
```

6. Getting test data and dropping irrelevant fields

```
#Getting the testing data
ts = pd.read_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/Date/date-2021-12-11.csv", index_col=0)

#Dropping irrelevent fields from test data
ts.drop(columns='date',inplace=True)
time = ts["time"]
time = time[0:-1]
ts.drop(columns='time',inplace=True)
ts.drop(columns='meal_type',inplace=True)
```

7. Pre-processing test data

```
#Preprocessing the test data
cgm = np.array(ts['cgm'])
bas = np.array(ts['bas'])
bol = np.array(ts['bol'])
meal_carb = np.array(ts['meal_carb'])
cgm = cgm.reshape((len(cgm),1))
bas = bas.reshape((len(bas), 1))
bol = bol.reshape((len(bol), 1))
meal_carb = meal_carb.reshape((len(meal_carb), 1))
ts = np.hstack((bas,bol,meal_carb,cgm))
cgm = cgm[1:]
ts = ts[0:-1]

shp = np.shape(ts)
ts = ts.reshape((shp[0], 1,  4,1))
```

8. Predicting data

```
#Predicting data
y_pred = model.predict(ts, verbose=0)
y_pred = y_pred.reshape(len(y_pred))
cgm = cgm.reshape(len(cgm))

y_pred = pd.DataFrame(list(y_pred))
cgm = pd.DataFrame(list(cgm))
print(y_pred, cgm)
```

Output:

```
               0
0    183.742798
1    186.640717
2    188.337372
3    189.881561
4    191.297363
..          ...
266   83.407707
267   83.407707
268   83.407707
269   83.407707
270   82.223358

[271 rows x 1 columns]        0
0    185
1    187
2    189
3    191
4    192
..   ...
266   84
267   84
268   84
269   83
270   84

[271 rows x 1 columns]
```

9. Performance metrics

```
#Performance metrics
mse = mean_squared_error(y_pred, cgm)
rmse = math.sqrt(mse)
print("Mean square error: ", mse)
print("Root mean square error: ", rmse)
```

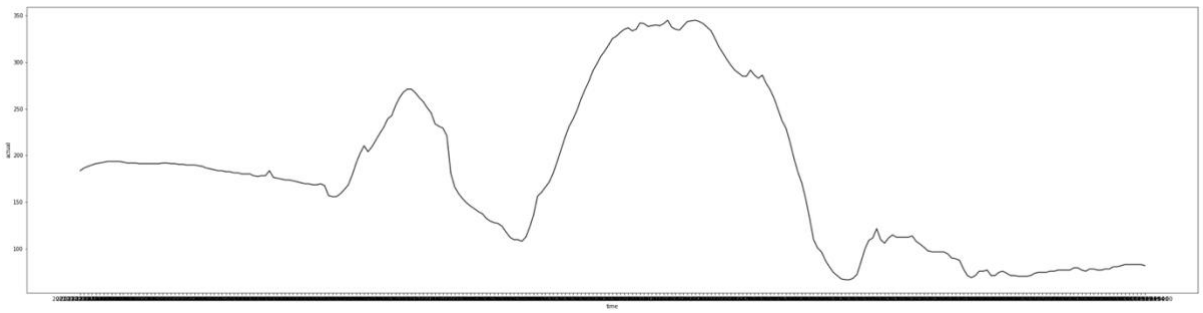
```
Mean square error:  57.03782852049324
Root mean square error:  7.552339274720995
```

10. Printing Predicted(Forecasted) vs Actual Data

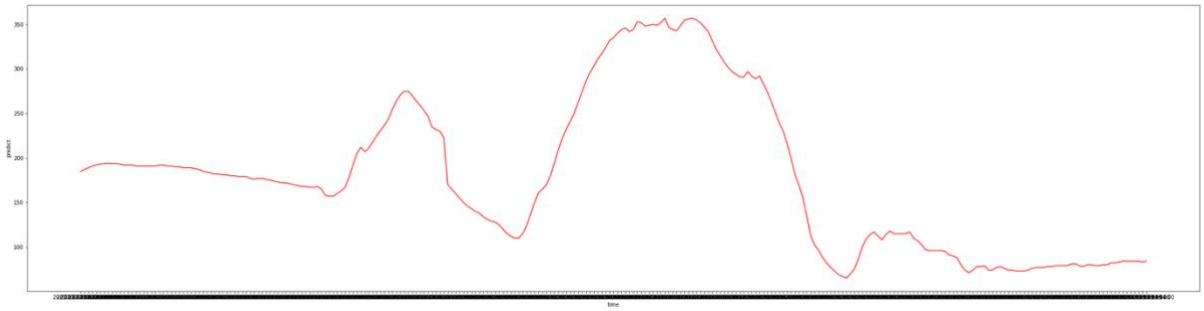
```
#Plotting graph
fig1, act = plt.subplots()
fig2, predict = plt.subplots()
fig1.set_figwidth(40)
fig1.set_figheight(10)
fig2.set_figwidth(40)
fig2.set_figheight(10)

act.plot(time, y_pred[0], color="black")
act.set_xlabel("time")
act.set_ylabel("actual")
predict.plot(time, cgm[0], color="red")
predict.set_xlabel("time")
predict.set_ylabel("predict")
```

Actual Data

```
Text(0, 0.5, 'predict')
```



Forecasted(Predicted) Data



6.5 INNOVATION

An attempt to segregate the meals consumed during the observed period into 2 clusters and identifying low GI and high GI foods based on heuristic graph analysis.

Pre-processing to include exercise data:

1. Reading the created .csv files

```
gl = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/gl.csv',index_col=0)
bas = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/bas.csv',index_col=0)
bol = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/bol.csv',index_col=0)
meal = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/meal.csv',index_col=0)

ex = pd.read_csv('/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/ex.csv', index_col=0)
```

2. Changing string to datetime object

```
#processing the time column: changing from string to datetime object
gl['time'] = pd.to_datetime(gl['time'],infer_datetime_format=True)
bas['time'] = pd.to_datetime(bas['time'],infer_datetime_format=True)
bol['time'] = pd.to_datetime(bol['time'],infer_datetime_format=True)
meal['time'] = pd.to_datetime(meal['time'],infer_datetime_format=True)

ex['time'] = pd.to_datetime(ex['time'],infer_datetime_format=True)
```

3. Extracting date from datetime object

```
#extracting date from time column to facilitate grouping of data according to date
gl['date'] = gl['time'].dt.date
bas['date'] = bas['time'].dt.date
bol['date'] = bol['time'].dt.date
meal['date'] = meal['time'].dt.date

ex['date'] = ex['time'].dt.date
```

4. Creating a new dataframe combining all the data into one table

```
#creating a new dataframe that acts as a template for combining all the patient data into one table
data = pd.DataFrame()
data['date'] = gl['date']
data['time'] = gl['time']
data['cgm'] = gl['value']
data['bas'] = [0]*len(data)
data['bol'] = [0]*len(data)
data['meal_carb'] = [0]*len(data)
data['meal_type'] = ["None"]*len(data)
data['ex'] = [0]*len(data)
```

5. Creating dataframe for each day's data

```
#creating dataframes for each day's data from the template dataframe
grouped_df = data.groupby(data.date)
keys = grouped_df.indices.keys()
keys = list(keys)

date_df = []
for i in range(len(keys)):
    df = grouped_df.get_group(keys[i])
    date_df.append(df)
```

6. Populating the dataframe with the CGM time as the reference

```
#populating the dataframes with data
for i in range(len(bas)):
    for k in range(len(date_df)):
        f = 0
        for j in range(len(date_df[k])-1):
            if bas.iloc[i,0]> date_df[k].iloc[j,1] and bas.iloc[i,0] <= date_df[k].iloc[j+1,1]:
                date_df[k].iloc[j+1,3] = bas.iloc[i,1]
                f = 1
                break
        if f==1:
            break

for i in range(len(bol)):
    for k in range(len(date_df)):
        f = 0
        if f==1:
            break
        for j in range(len(date_df[k])-1):
            if bol.iloc[i,0]> date_df[k].iloc[j,1] and bol.iloc[i,0] <= date_df[k].iloc[j+1,1]:
                date_df[k].iloc[j+1,4] = bol.iloc[i,1]
                f = 1
                break
        if f==1:
            break

for i in range(len(meal)):
    for k in range(len(date_df)):
        f = 0
        if f==1:
            break
        for j in range(len(date_df[k])-1):
            if meal.iloc[i,1]> date_df[k].iloc[j,1] and meal.iloc[i,1] <= date_df[k].iloc[j+1,1]:
                date_df[k].iloc[j+1,5] = meal.iloc[i,2]
                date_df[k].iloc[j+1,6] = meal.iloc[i,0]
                f = 1
                break
        if f==1:
            break
```

```
for i in range(len(ex)):
    for k in range(len(date_df)):
        f = 0
        if f==1:
            break
        for j in range(len(date_df[k])-1):
            if ex.iloc[i,0]> date_df[k].iloc[j,1] and ex.iloc[i,0] <= date_df[k].iloc[j+1,1]:
                for l in range(int(ex.iloc[i,1]/5)):
                    date_df[k].iloc[j+l+1,7] = ex.iloc[i,2]
                f = 1
                break
        if f==1:
            break
```

7. Creating csv file for each day's record

```
#creating csv file for each day's record
for i in range(len(keys)):
    string = "date-" + str(keys[i])
    date_df[i].to_csv("/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/Date/"+string+".csv")
```

Sample data frame:

	date	time	cgm	bas	bol	meal_carb	meal_type	ex
0	2021-09-13	2021-09-13 12:33:00	219	0.0	0.0	0	None	0
1	2021-09-13	2021-09-13 12:38:00	229	0.0	0.0	0	None	0
2	2021-09-13	2021-09-13 12:43:00	224	0.0	0.0	0	None	0
3	2021-09-13	2021-09-13 12:48:00	221	0.0	0.0	0	None	0
4	2021-09-13	2021-09-13 12:53:00	215	0.0	0.0	0	None	0
...	...	...	...	...	...	...	...	...
130	2021-09-13	2021-09-13 23:38:00	132	0.0	0.0	0	None	0
131	2021-09-13	2021-09-13 23:43:00	128	0.0	0.0	0	None	0
132	2021-09-13	2021-09-13 23:48:00	126	0.0	0.0	0	None	0
133	2021-09-13	2021-09-13 23:53:00	125	0.0	0.0	0	None	0
134	2021-09-13	2021-09-13 23:58:00	124	0.0	0.0	0	None	0

135 rows × 8 columns

Training model to cluster meal data:

1. Imports

```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from tslearn.clustering import TimeSeriesKMeans
from tslearn.utils import to_time_series_dataset
```

2. Pre-processing data from all files into a single list of meal data

```
meal = []
import os
path = "/Users/shreyaananth/Desktop/College/CIP/Code/Data/563/Date"
for file in os.listdir(path):
    full_path = os.path.join(path,file)
    df = pd.read_csv(full_path, index_col=0)
    df.drop(columns='date',inplace=True)
    df.drop(columns='time',inplace=True)
    df.drop(columns='meal_type',inplace=True)

    for i in range(len(df)):
        if df.iloc[i,3]!=0:
            temp = []
            j = i
            while True:
                if j+1==len(df) or df.iloc[j,1]!=0 or df.iloc[j,2]!=0 or df.iloc[j,4]!=0:
                    break
                temp.append(df.iloc[j+1,0]-df.iloc[j,0])
                j = j+1
            if len(temp)>5 and len(temp)<=30:
                meal.append(temp)
```

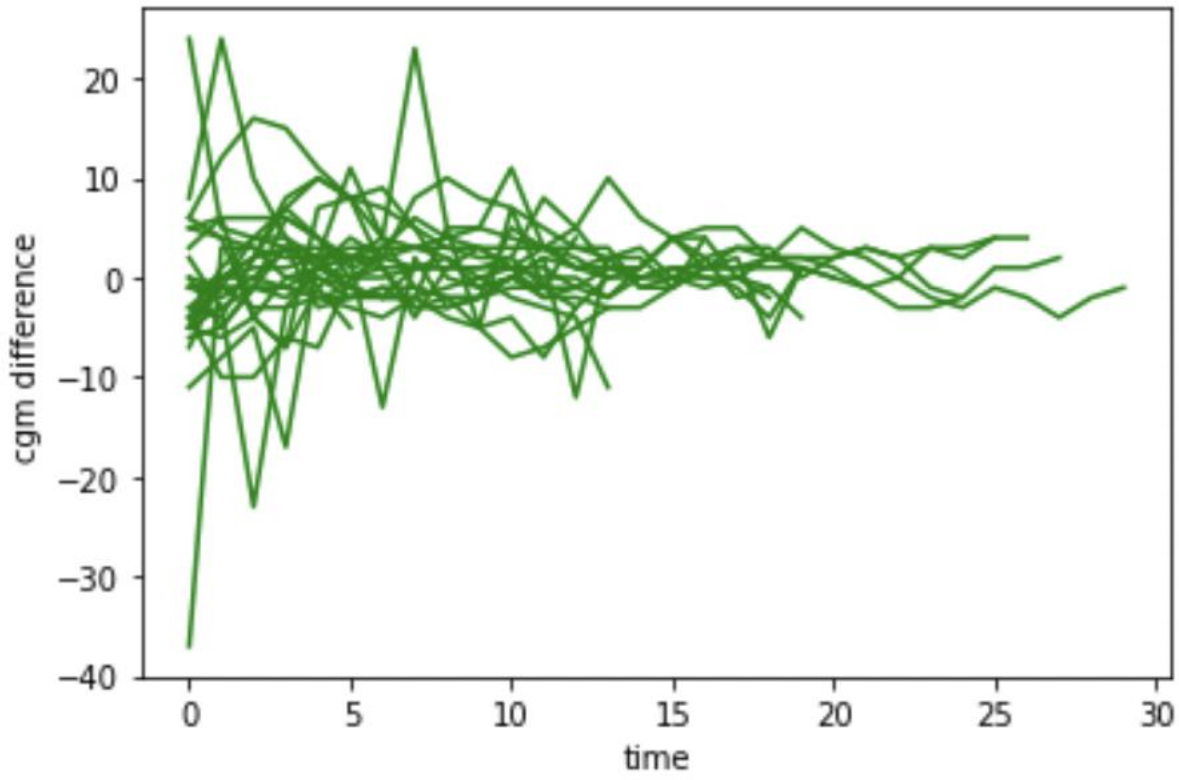
3. Segregating the datapoints into 2 clusters using time series KMeans

```
X = to_time_series_dataset(meal)
model = TimeSeriesKMeans(n_clusters=2, metric="dtw", max_iter=10, random_state=42)
labels = model.fit_predict(X)
```

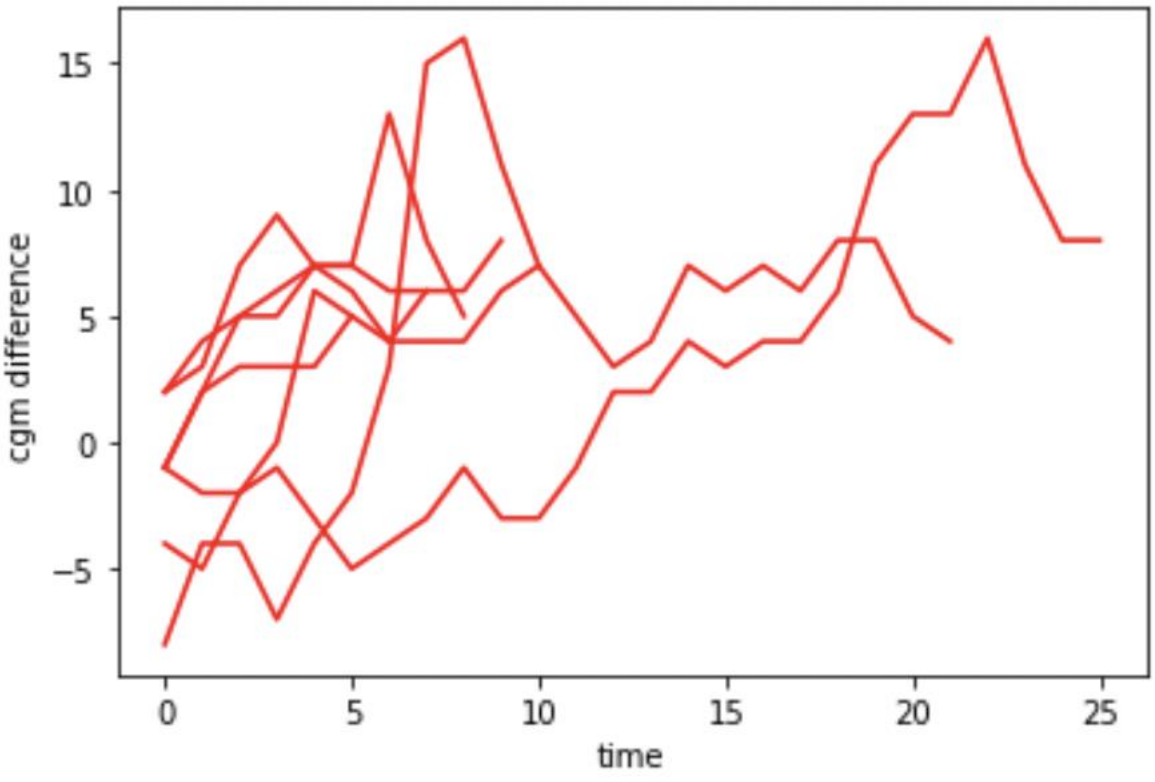
4. Plotting the meal data (which are small time series) as part of their respective clusters

```
fig, one = plt.subplots()
one.set_xlabel("time")
one.set_ylabel("cgm difference")

fig, two = plt.subplots()
two.set_xlabel("time")
two.set_ylabel("cgm difference")

for i in range(len(meal)):
    time = list(range(len(meal[i])))
    if labels[i]==1:
        two.plot(time, meal[i], color = "red")
    else:
        one.plot(time, meal[i], color = "green")
```

Cluster 1:



Cluster 2:



From the above results we conclude the following:

Since the 1st plot has classified data, whose trend is not sharp but flattened, and because they are centred around 0 (no observed difference between CGM value of 2 consecutive observations), we can conclude that the meal items considered by this cluster belong to the **Low GI class** of foods.

In contrast the 2nd plot shows data that have visible spikes and increase in the difference between consecutive CGM value. Thus, we can conclude that the meal items considered by this cluster belong to the **High GI class** of foods.

7. RESULTS AND DISCUSSIONS

7.1 SARIMAX MODEL FOR UNIVARIATE TIME-SERIES PREDICTION

The SARIMAX Model is for forecasting the future values of a particular curve based on past values of the given time-series. Four kinds of components help make a time series, and also they can affect our time series analysis if present in excess. So here, for this time series, we need to check more for the availability of components. The components are observed, trend, seasonal and residual values.

To perform forecasting using the ARIMA model, we required a stationary time series. Stationary time series is a time series that is unaffected by these four components. Most often, it happens when the data is non-stationary the predictions we get from the ARIMA model are worse or not that accurate.

Prediction:

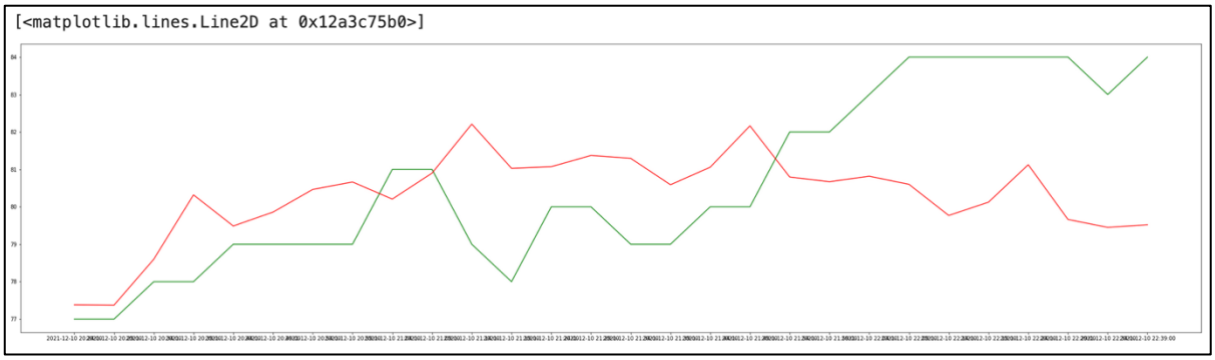
```
model = SARIMAX(training['cgm'], order=(5,1,1), seasonal_order=(1,1,1,7))
model_fit = model.fit()
fc = model_fit.forecast(len(data)-trainlen, alpha = 0.1).reset_index()
fc.drop(columns='index', inplace=True)
mse = mean_squared_error(fc, testing['cgm'])
print('Actual\tPredicted')
for i in range(trainlen,len(data)):
    print(testing['cgm'][i], fc['predicted_mean'][i-trainlen])
```

Best Case Scenario:

```
import math
print('Value after 15 min (Actual,predicted): ',testing['cgm'][trainlen+2], fc['predicted_mean'][2])
print('Value after 60 min (Actual,predicted): ',testing['cgm'][trainlen+11], fc['predicted_mean'][11])
print('Root mean squared error: ',math.sqrt(mse))
```

```
Value after 15 min (Actual,predicted):  78 78.59903866463145
Value after 60 min (Actual,predicted):  78 81.02828565265654
Root mean squared error:  2.397020965539997
```

```
[<matplotlib.lines.Line2D at 0x12a3c75b0>]
```



Green – Actual

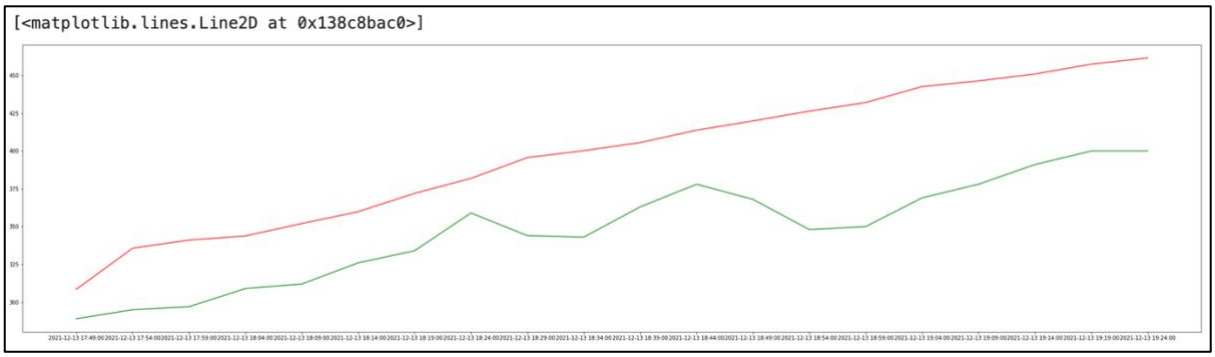
Red – Predicted

Worst Case Scenario:

```
import math
print('Value after 15 min (Actual,predicted): ',testing['cgm'][trainlen+2], fc['predicted_mean'][2])
print('Value after 60 min (Actual,predicted): ',testing['cgm'][trainlen+11], fc['predicted_mean'][11])
print('Root mean squared error: ',math.sqrt(mse))
```

```
Value after 15 min (Actual,predicted):  297 341.08282344526725
Value after 60 min (Actual,predicted):  378 413.7239129048619
Root mean squared error:  52.54350517446053
```

```
[<matplotlib.lines.Line2D at 0x138c8bac0>]
```



Green – Actual

Red – Predicted

Advantages:

- Predicts future values with a small set of training data.
- Gives good results even for non-stationary time series.

Disadvantages:

- Doesn't take into account multiple features.
- Error metrics keep on increasing if we try to predict far in the future.
- Doesn't create a generalized model that can be used to test all datasets.

7.2 LSTM MODEL FOR UNIVARIATE TIME-SERIES PREDICTION

LSTM (Long Short-Term Memory) is a Recurrent Neural Network (RNN) based architecture that is widely used in natural language processing and time series forecasting. The LSTM rectifies a huge issue that recurrent neural networks suffer from: short-memory. Using a series of 'gates,' each with its own RNN, the LSTM manages to keep, forget or ignore data points based on a probabilistic model.

LSTMs also help solve exploding and vanishing gradient problems. While exploding and vanishing gradients are huge downsides of using traditional RNN's, LSTM architecture severely mitigates these issues. After a prediction is made, it is fed back into the model to predict the next value in the sequence.

Data:

```
data = pd.read_csv('Data-for-Review/date-2021-12-13.csv', usecols=['time','cgm'])
dataset = ['Data-for-Review/date-2021-12-07.csv','Data-for-Review/date-2021-12-08.csv',
           'Data-for-Review/date-2021-12-09.csv','Data-for-Review/date-2021-12-10.csv',
           'Data-for-Review/date-2021-12-11.csv','Data-for-Review/date-2021-12-12.csv',
           'Data-for-Review/date-2021-12-13.csv']
df = []
for d in dataset:
    t = pd.read_csv(d,usecols=['time','cgm'])
    df.append(t)
```

Prediction:

```
#LSTM model to predict the glucose level
#look_back = 1

from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

p = data.iloc[:-1, 1]
q = data.iloc[1:,1]

model1 = Sequential()

model1.add(LSTM(units=50, return_sequences=True, input_shape=(1, 1)))
model1.add(Dropout(0.2))

model1.add(LSTM(units=50, return_sequences=True))
model1.add(Dropout(0.2))

model1.add(LSTM(units=50, return_sequences=True))
model1.add(Dropout(0.2))

model1.add(LSTM(units=50))
model1.add(Dropout(0.2))

model1.add(Dense(units = 1))
model1.compile(optimizer = 'adam', loss = 'mean_squared_error')

for i in range(350):
    print(f'Epoch - {i}')
    for d in df:
        p = d.iloc[:-1,1]
        q = d.iloc[1:, 1]
        model1.fit(p, q, epochs = 1, batch_size = 32)
```

```
data1 = pd.read_csv('Data-for-Review/date-2021-12-29.csv', usecols=['time','cgm'])
p = data1.iloc[:-1, 1]
q = data1.iloc[1:, 1]
predictions = model1.predict(p)
```

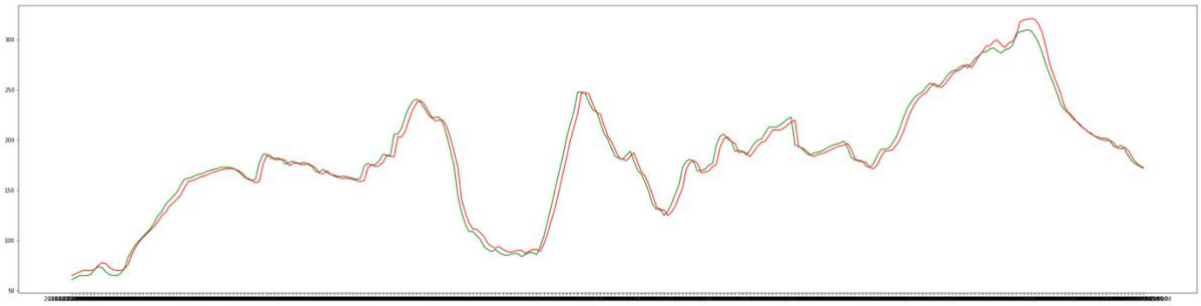
```
pred = []
for i in range(len(predictions)):
    pred.append(predictions[i][0])
print("Actual Predicted")
for i in range(len(pred)):
    print(q[i+1], pred[i])
```

Output:

```
mse = mean_squared_error(pred, q)
print(math.sqrt(mse))
```

```
7.259013631424979
```

```
[<matplotlib.lines.Line2D at 0x13889c7c0>]
```



Green – Actual

Red – Predicted

Difference between LSTM and SARIMAX:

- ARIMA (and MA-based models in general) are designed for time series data while RNN-based models are designed for sequence data. Because of this distinction, it's harder to build RNN-based models out-of-the-box.
- ARIMA models are highly parameterized and due to this, they don't generalize well. Using a parameterized ARIMA on a new dataset may not return accurate results. RNN-based models are non-parametric and are more generalizable.

So, as a whole, LSTM is a better model for Univariate time-series prediction.

7.3 CRNN MODEL FOR MULTIVARIATE TIME-SERIES PREDICTION

Data:

```
#Preprocessing the dataframe for training after extracting it from csv file
datasets = []
cgm_data = []
import os
dataset = ['Data-for-Review/date-2021-12-07.csv','Data-for-Review/date-2021-12-08.csv',
           'Data-for-Review/date-2021-12-09.csv','Data-for-Review/date-2021-12-10.csv',
           'Data-for-Review/date-2021-12-11.csv','Data-for-Review/date-2021-12-12.csv',
           'Data-for-Review/date-2021-12-13.csv']
```

Prediction:

```
#Creating CRNN Model
model = Sequential()
model.add(Conv2D(8, kernel_size=(12,12), padding='same',activation="relu", input_shape = (1,4,1)))
model.add(MaxPooling2D(1))
model.add(Conv2D(7, kernel_size=(16,6), padding='same',activation="relu"))
model.add(MaxPooling2D(1))
model.add(Conv2D(5, kernel_size=(8,8), padding='same',activation="relu"))
model.add(MaxPooling2D(1))
model.add(Dropout(0.2))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(68, return_sequences=True))
model.add(Dropout(0.2))
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

```
#Getting the testing data
ts = pd.read_csv("Data-for-Review/date-2021-12-21.csv", index_col=0)
```

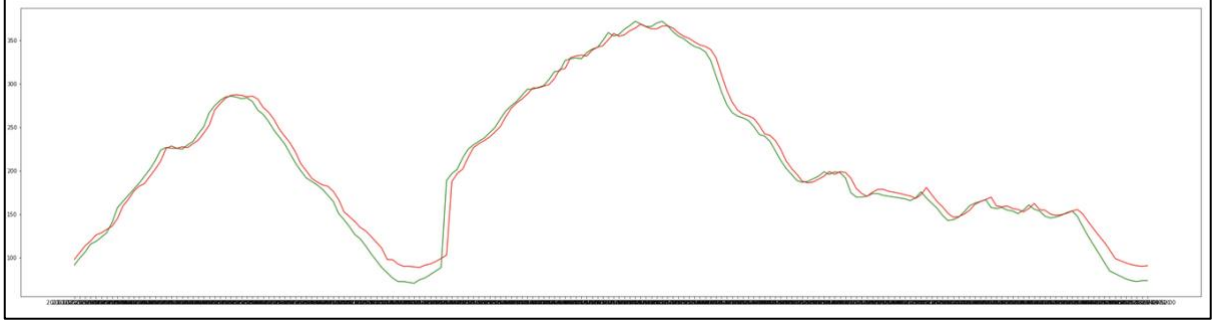
```
#Dropping irrelevent fields from test data
ts.drop(columns='date',inplace=True)
time = ts["time"]
time = time[0:-1]
ts.drop(columns='time',inplace=True)
ts.drop(columns='meal_type',inplace=True)
```

Output:

```
#Performance metrics
mse = mean_squared_error(y_pred, cgm)
rmse = math.sqrt(mse)
print("Mean square error: ", mse)
print("Root mean square error: ", rmse)
```

```
Mean square error:  126.09518908537837
Root mean square error:  11.229211418678444
```

```
[<matplotlib.lines.Line2D at 0x136aecbb0>]
```



Green – Actual

Red – Predicted

8. CONCLUSION AND FUTURE WORKS:

Therefore we have devised a strategy to predict future glucose levels in the presence and absence of external factors. The distinct boundaries between the algorithms used for univariate time series prediction is observed. The efficiency of Multi-variate time series prediction is calculated over a real-time dataset. A timeseries clustering was done to cluster foods as low-Gi or high-Gi.

Future works might include taking into consideration, another extensive set of external factors, so that algorithm is as generalised as possible and for more accurate GI classification.

9. REFERENCES

[1] T. Hamdi et al., "Artificial neural network for blood glucose level prediction," in 2017 International Conference on Smart, Monitored and Controlled Cities (SM2C), 2017.

[2] K. Li, J. Daniels, C. Liu, P. Herrero, and P. Georgiou, "Convolutional recurrent neural networks for glucose prediction," IEEE J. Biomed. Health Inform., vol. 24, no. 2, pp. 603–613, 2020.

[3] S. Mirshekarian, R. Bunescu, C. Marling, and F. Schwartz, "Using LSTMs to learn physiological models of blood glucose behavior," Annu Int Conf IEEE Eng Med Biol Soc, vol. 2017, pp. 2887–2891, 2017.

[4] W. P. T. M. van Doorn et al., "Machine learning-based glucose prediction with use of continuous glucose and physical activity monitoring data: The Maastricht Study," PLoS One, vol. 16, no. 6, p. e0253125, 2021.

[5] S. Mirshekarian, H. Shen, R. Bunescu, and C. Marling, "LSTMs and neural attention models for blood glucose prediction: Comparative experiments on real and synthetic data," Annu Int Conf IEEE Eng Med Biol Soc, vol. 2019, pp. 706–712, 2019.