Enhancing Patient Care: A Real-Time Healthcare Monitoring, Alerting, And Predictive System

—

Group 8

Anshit Verma, Manvi Goel, Navdha Agarwal, Padma Priya Mukkamala, Reshma Ramachandra, Sindhu Raghuram Panyam

Project Overview

Revolutionize healthcare monitoring through a proactive and personalized healthcare monitoring system capable to handle large data

- Develop predictive models like ARIMA, Prophet and XGBoost to forecast health conditions
- Utilize real-time big data analysis system (Kafka, Flink) to tailor alerts based on individual patient profiles
- Compare predicted behavior with real-time alerts to evaluate the effectiveness of our predictive analytics in improving patient outcomes.

Dataset

- The PhysioNet/Computing in Cardiology Challenge 2012 dataset is used.
- We focus on the heart rate and respiratory rate metrics of ICU Patients.
- The dataset contains varying metrics for patients - some have only heart rate (HR), some have only respiratory rate (RespRate), while some have both or neither.
- Preprocessing involves identification of patients with both heart rate and respiratory rate data.
- To ensure uniform timestamps across all patients, we drop certain times or interpolate data for consistency.
- To expand the dataset we use interpolation to generate 2900 data points per patient for increased statistical robustness.

Dataset

Train-Test Split: Train set consists of 2800 data points per patient, while the test set comprises 100 data points per patient.

Train set examples

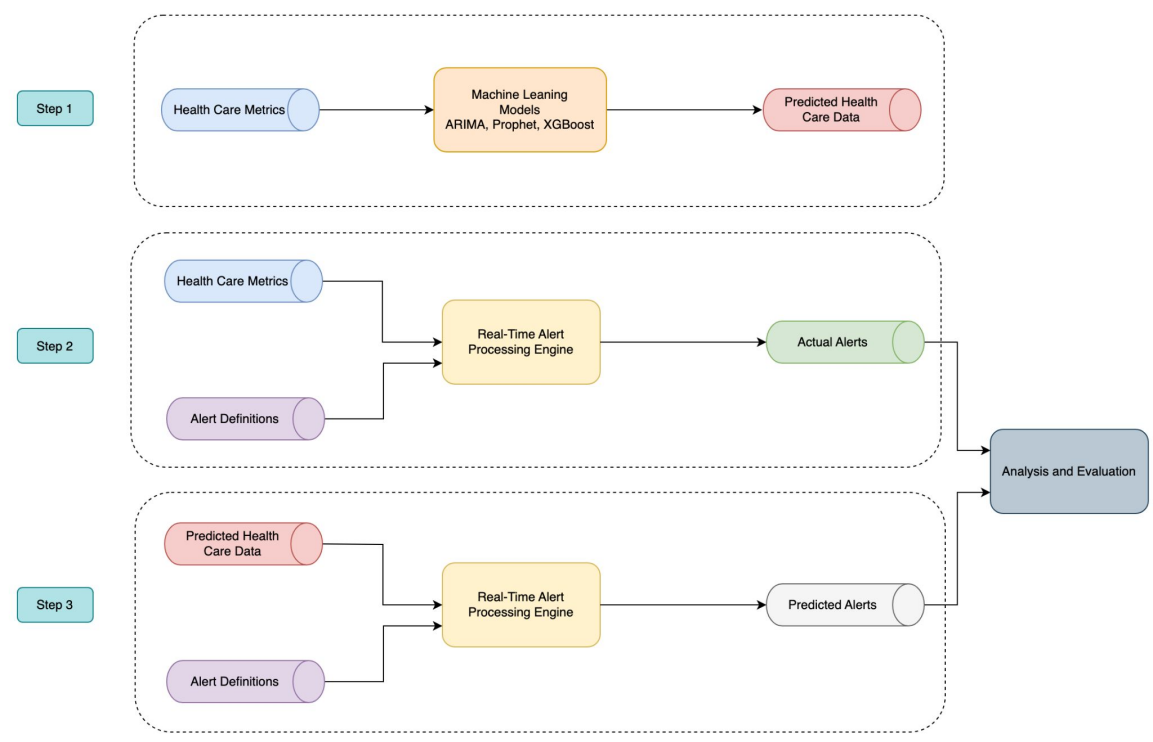
	PatientID	HR	RespRate	Timestamp
23208	133696	85.0	15.0	1
23209	133696	85.0	15.0	2
23210	133696	85.0	15.0	3
23211	133696	85.0	15.0	4
23212	133696	85.0	15.0	5

Test set examples

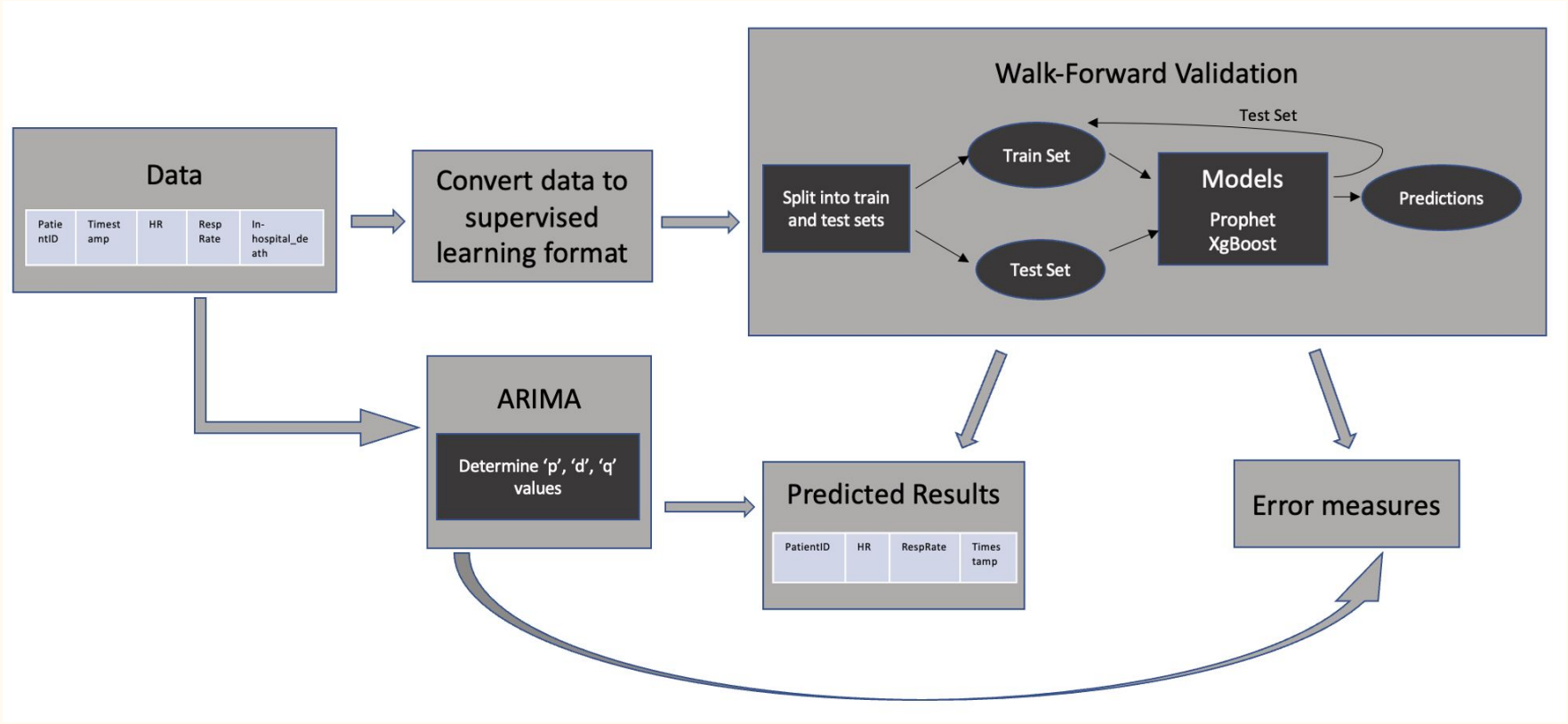
	PatientID	Timestamp	HR	RespRate
63817	133696	2897	83.0	21.0
63818	133696	2898	83.0	21.0
63819	133696	2899	83.0	21.0
63820	133696	2900	83.0	21.0
63821	133696	2901	83.0	21.0

SYSTEM ARCHITECTURE

System Architecture



Predictive Analytics



CODE WALKTHROUGH

Dataset Creation

- We process text files from the dataset of PhysioNet ICU Challenge 2012, extracting heartbeat and respiratory rate-related data for each patient.
- We consolidate readings of 40 consecutive timestamps for individual patients, organizing this information into a CSV file format for further processing.
- We then generate new interpolated data using NumPy's np.interp() function based on a defined time range, and then organize the interpolated data into a new file.

```
for root, dirs, files in os.walk(data_dir):
    for file in files:
        if file.endswith('.txt'):
            patient_timestamps=[]
            pateint_data={}
            t1 = 0
            record_name = os.path.splitext(file)[0]
            record_data=open(data_dir+"/"+file)
            for line in record_data:
                if "00:00," in line:
                    continue
                elif "Time," in line:
                    continue
                elif ",RespRate," in line:
                    if t1==0:
                        t1=line.split(",")[0].replace(":",".")
                    timestamp=str(round(float(line.split(",")[0].replace(":",".")) - float(t1),0))
                    patient_timestamps.append(timestamp)
                    pateint_data[timestamp]=record_name+","+timestamp+","+line.split(",")[1]+","+line.split(",")[2]
```

```
new_time = np.arange(data['Timestamp'].min(), data['Timestamp'].max() + 0.01, 0.01)

interpolated_data = pd.DataFrame(columns=['PatientID', 'Timestamp', 'Metric', 'Value'])

for (patient_id, metric), group in data.groupby(['PatientID', 'Metric']):
    interpolated_values = np.interp(new_time, group['Timestamp'], group['Value'])

    interpolated_values = np.round(interpolated_values, 0)
    new_time_rounded = np.round(new_time, 2)

    new_rows = pd.DataFrame({
        'PatientID': patient_id,
        'Timestamp': new_time_rounded,
        'Metric': metric,
        'Value': interpolated_values
    })

    interpolated_data = pd.concat([interpolated_data, new_rows])

interpolated_data.to_csv("RespRate_interpolated_data.csv", index=False)
```

ARIMA

- We convert the timeseries to a supervised pandas series for the model with independent and dependent variables.
- We then select parameters on the basis of data analysis done. Appropriate lag, differencing degree and moving average window is selected as p, d, q respectively. The P, D , Q parameters are for the seasonality.
- We then forecast with the model for the next 100 timesteps after the model is trained and create the final dataframe.

```
#Dataframe
patient_data = data[data.PatientID == 142580]
patient_data_series =  patient_data.Value.reset_index(drop=True)

#Set initial values and some bounds
ps = range(0, 5)
d = 1
qs = range(0, 5)
Ps = range(0, 5)
D = 1
Qs = range(0, 5)
s = 5

#Create a list with all possible combinations of parameters
parameters = product(ps, qs, Ps, Qs)
parameters_list = list(parameters)
len(parameters_list)

#Create an ARIMA model with the above defined parameters and difference (lag) of 1
model = sm.tsa.statespace.SARIMAX(patient_data_series,
                                  order=(0, d, 0),
                                  seasonal_order=(2, D, 4, s)).fit(disp=-1)

#Computing the model AIC
model.aic

#Forecasting with the trained model for next 100 timesteps
forecast = model.forecast(100)

#Combining the predictions with the original data
final_df = pd.DataFrame({'PatientID': 142580, 'HR': pd.concat(
  [patient_data_series, forecast])})
final_df = final_df.reset_index()
final_df = final_df.rename(columns = {'index': 'Timestamp'})

#Calculating Mean Absolute Error
error = mean_absolute_error(test_data_series, forecast)
print(f'MAE = {error}')
```

XGBoost

- First, we transform the dataset into a supervised learning format.
- Each input-output pair consists of the input of historical time steps, and the target variable to be predicted in the future time step as the output.
- The xgboost_forecast function trains an XGBoost regressor using historical data.
- This splits the input data into features (input columns) and target variable (output column), fits the XGBoost model, and makes a one-step prediction for the test set.

```
# transform a time series dataset into a supervised learning dataset
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
  n_vars = 1 if type(data) is list else data.shape[1]
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  if dropnan:
    agg.dropna(inplace=True)
  return agg.values

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test, :], data[-n_test:, :]

# fit an xgboost model and make a one step prediction
def xgboost_forecast(train, testX):
  # transform list into array
  train = asarray(train)
  # split into input and output columns
  trainX, trainy = train[:, :-1], train[:, -1]
  # fit model
  model = XGBRegressor(objective='reg:squarederror', n_estimators=1000)
  model.fit(trainX, trainy)
  # make a one-step prediction
  yhat = model.predict(asarray([testX]))
  return yhat[0]
```

XGBoost implementation

XGBoost

- The overall time series analysis uses a walk-forward validation approach.
- It iteratively goes through the test set, updating the model at each step with the latest observed data, and making predictions for the subsequent time step.
- The performance of the model is evaluated using the Mean Absolute Error (MAE) between the predicted and actual values.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # split test row into input and output columns
    testX, testy = test[i, :-1], test[i, -1]
    # fit model on history and make a prediction
    yhat = xgboost_forecast(history, testX)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
    # summarize progress
    #print('>expected=%.1f, predicted=%.1f' % (testy, yhat))
  # estimate prediction error
  error = mean_absolute_error(test[:, -1], predictions)
  return error, test[:, -1], predictions
```

XGBoost implementation

PROPHET

- We transform the dataset into a supervised learning format with, each input-output pair consists of the input of historical time steps in the form of ds (datetime) and y (target to be predicted in the future) columns.
- The prophet_forecast function trains an instance of the Prophet model using historical data.
- The walk-forward validation remains the same as the approach followed for Xgboost.

```python
def series_to_supervised(df, patient_id, n_in=1, n_out=1, dropnan=True):
    # Filter data for the specified patient ID
    patient_data = df[df['PatientID'] == patient_id][['PatientID', 'Timestamp', 'HR']].copy()

    # Prepare the DataFrame for Prophet with 'ds' and 'y' columns
    supervised_df = pd.DataFrame()
    supervised_df['ds'] = patient_data['Timestamp']
    supervised_df['y'] = patient_data['HR']

    if dropnan:
        supervised_df.dropna(inplace=True)

    print("Patient Data:")
    print(patient_data.head())

    return supervised_df
```

```python
def train_test_split(data, n_test):
    return data.iloc[:-n_test], data.iloc[-n_test:]
```

```python
def prophet_forecast(train, testX):
    model = Prophet()
    model.fit(train)
    future = pd.DataFrame({'ds': testX['ds']})
    forecast = model.predict(future)
    yhat = forecast['yhat'].values
    return yhat
```

```python
def walk_forward_validation(data, n_test):
    predictions = []
    train, test = train_test_split(data, n_test)
    history = train.copy()

    for i in range(len(test)):
        testX = test.iloc[[i]]
        yhat = prophet_forecast(history, testX)
        predictions.append(yhat[0])
        history = history.append(testX)

    error = mean_absolute_error(test['y'].values, predictions)
    return error, test['y'].values, predictions
```

Real Time System

Data (Body vitals, and alert definition) read from CSV and pushed to Kafka queues

Step 1: Reading vitals from .csv files, and storing them in a HashMap

```
public Map<String, List<VitalDTO>> getVitalData(List<String> patientIds) throws Exception {
    Map<String, List<VitalDTO>> vitalDataset = new HashMap<>();
    LocalDateTime startTime = LocalDateTime.now();
    BufferedReader br = new BufferedReader(new FileReader( fileName: "./dataset/results/prophet.csv"));
    br.readLine();
    String line;

    int count = 0;
    while ((line = br.readLine()) != null && count++ < 10000) {
        String[] values = line.split( regex: ",");
        String patientId = values[0];
        long seconds = (long) (Float.parseFloat(values[1]));
        double heartRateValue = Double.parseDouble(values[2]);
        heartRateValue = Math.round(heartRateValue);
        String time = startTime.plusSeconds(seconds).format(TimeConversion.formatter);

        if (patientIds.contains(patientId)) {
            VitalDTO vitalHRDTO = new VitalDTO().builder()
                    .time(time)
                    .patientId(patientId)
                    .vitalValue(heartRateValue)
                    .vitalName("HeartRate").build();
            vitalHRDTO.calculateKey();
            vitalDataset.putIfAbsent(patientId, new ArrayList<>());
            vitalDataset.get(patientId).add(vitalHRDTO);
        }
    }
    return vitalDataset;
}
```

Step 2: Scheduler - sending vitals data to “vitals” Kafka queue every 1 sec

```
@Scheduled(fixedRate = 1000)
public void generateVitalEvents() {
    patientIds.forEach(patientId -> {
        vitalDTOKafkaProducer.sendObjectToKafka(dataHR.get(patientId).get(index),
                appConfig.getVitalKafkaTopic());
    });
    index++;
}
```

Step 3: REST API - sending Alert Definition to Kafka queue using a Postman Request

```
@PostMapping("/add")
public ResponseEntity<AlertDefinitionDTO> createAlertDefinition(
        @RequestBody AlertDefinitionDTO alertDefinition) {

    String alertId = UUID.randomUUID().toString();
    alertDefinition.setAlertID(alertId);
    alertDefinition.calculateKey();

    alertDefinitionDTOKafkaProducer.sendObjectToKafka(alertDefinition,
            appConfig.getAlertDefinitionKafkaTopic());
    return ResponseEntity.ok(alertDefinition);
}
```

Real Time System

Processing Alerts in Apache Flink Application

Step 1: Kafka Sources for Flink

```
public void processAlerts() throws Exception {

    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    KafkaSource<VitalDTO> vitalSource = KafkaSource.<VitalDTO>builder()
            .setBootstrapServers(appConfig.getKafkaBootstrapServers())
            .setTopics(appConfig.getVitalKafkaTopic())
            .setGroupId(appConfig.getKafkaGroupId())
            .setStartingOffsets(OffsetsInitializer.latest())
            .setValueOnlyDeserializer(vitalEventDeserializationSchema)
            .build();

    KafkaSource<AlertDefinitionDTO> alertSource = KafkaSource.<AlertDefinitionDTO>builder()
            .setBootstrapServers(appConfig.getKafkaBootstrapServers())
            .setTopics(appConfig.getAlertDefinitionKafkaTopic())
            .setGroupId(appConfig.getKafkaGroupId())
            .setStartingOffsets(OffsetsInitializer.latest())
            .setValueOnlyDeserializer(alertEventDeserializationSchema)
            .build();

    DataStream<VitalDTO> vitalDTODataStream = env.fromSource(vitalSource, customWatermarkStrategy,
            sourceName: "Vital Kafka Source", TypeInformation.of(VitalDTO.class));
    DataStream<AlertDefinitionDTO> alertDTODataStream = env.fromSource(alertSource,
            customWatermarkStrategyAlert, sourceName: "Alert Kafka Source",
            TypeInformation.of(AlertDefinitionDTO.class));
```

Step 2: Alert Processing Module

```
ConnectedStreams<VitalDTO, AlertDefinitionDTO> connectedStreams = vitalDTODataStream
        .connect(alertDTODataStream)
        .keyBy(VitalDTO::getKey, AlertDefinitionDTO::getKey);

connectedStreams.process(new CustomKeyedCoProcessFunction(Time.seconds(5), Time.seconds(1)))
        .sinkTo(kafkaSink());
```

Step 3: Sinking generated Alerts to Kafka Topic

```
private KafkaSink<AlertDTO> kafkaSink() {
    KafkaRecordSerializationSchema<AlertDTO> kafkaRecordSerializationSchema =
            KafkaRecordSerializationSchema.builder()
                    .setTopic(appConfig.getAlertProducerKafkaTopic())
                    .setValueSerializationSchema(alertSerializationSchema).build();

    return KafkaSink.<AlertDTO>builder()
            .setBootstrapServers(appConfig.getKafkaBootstrapServers())
            .setRecordSerializer(kafkaRecordSerializationSchema)
            .setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_ONCE)
            .build();
```

Real Time System

Processing incoming Events, triggering Windows, and generating final Alerts

Step 1: Processing incoming Vital Events and generating Windows

```
public void processElement1(VitalDTO vitalDTO, KeyedCoProcessFunction<String, VitalDTO,
        AlertDefinitionDTO, AlertDTO>.Context context, Collector<AlertDTO> collector)
        throws Exception {

    Window currWindow = createWindow(context.timerService().currentWatermark());
    windowMapState.put(currWindow.getEndTime(), currWindow);
    context.timerService().registerEventTimeTimer(currWindow.getEndTime());

    windowMapState.values().forEach(window -> {
        long timeOfEvent = TimeConversion.stringToEpochMilli(vitalDTO.getTime());
        if (window.getStartTime() <= timeOfEvent && timeOfEvent <= window.getEndTime()) {
            window.getVitalDTOList().add(vitalDTO);
            window.getValues().add(vitalDTO.getVitalValue());
        }
    });
}
```

Step 2: Processing Alert Definition Events

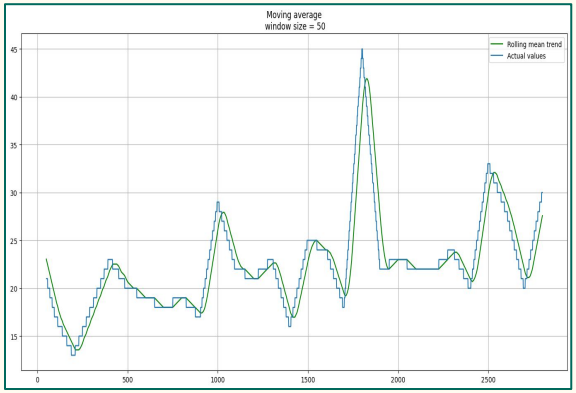
```
@Override
public void processElement2(AlertDefinitionDTO alertDefinitionDTO, KeyedCoProcessFunction<String,
        VitalDTO, AlertDefinitionDTO, AlertDTO>.Context context, Collector<AlertDTO> collector)
        throws Exception {

    alertDefinitionDTOMapState.put(alertDefinitionDTO.getAlertID(), alertDefinitionDTO);
}
```

Step 3: Applying Alert Definitions on Windows and generating final Alerts

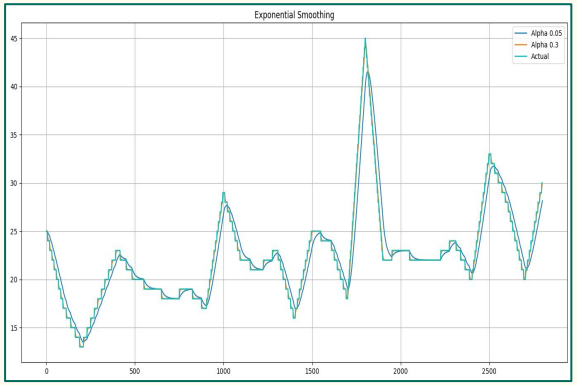
```
@Override
public void onTimer(long ts, KeyedCoProcessFunction<String, VitalDTO,
        AlertDefinitionDTO, AlertDTO>.OnTimerContext context, Collector<AlertDTO> collector)
        throws Exception {

    long timestampOfTrigger = context.timestamp();
    log.debug("Timer triggered for window ending at :: {}",
            TimeConversion.epochMilliToLocalDateTime(timestampOfTrigger));
    Window timedOutWindow = windowMapState.get(timestampOfTrigger);
    alertDefinitionDTOMapState.values().forEach(alertDefinitionDTO -> {
        log.debug("Values are:: {}", timedOutWindow.getValues());
        boolean alert =
                timedOutWindow.getValues().stream().allMatch(value ->
                        value >= alertDefinitionDTO.getAlertThreshold());
        if (alert) {
            log.info("ALERT for window {} - {}",
                    TimeConversion.epochMilliToLocalDateTime(timedOutWindow.getStartTime()),
                    TimeConversion.epochMilliToLocalDateTime(timedOutWindow.getEndTime()));
            collector.collect(createAlert(alertDefinitionDTO, timedOutWindow));
        }
    });
    windowMapState.remove(timestampOfTrigger);
}
```

EXPERIMENTS & VISUAL EXAMPLES

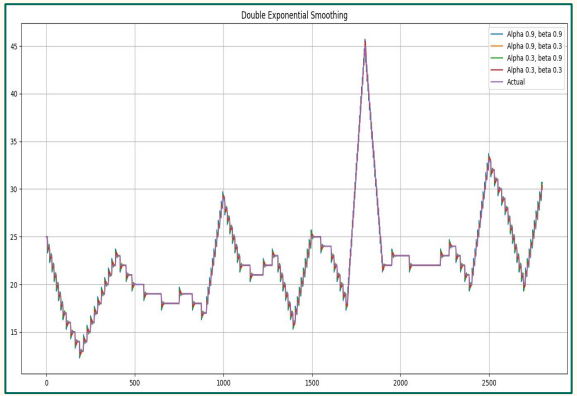
Analysis of Time Series Data



Moving average can be used to identify interesting trends in the data. We define a 50 time period window to apply the moving average model to smooth the time series, and highlight different trends.

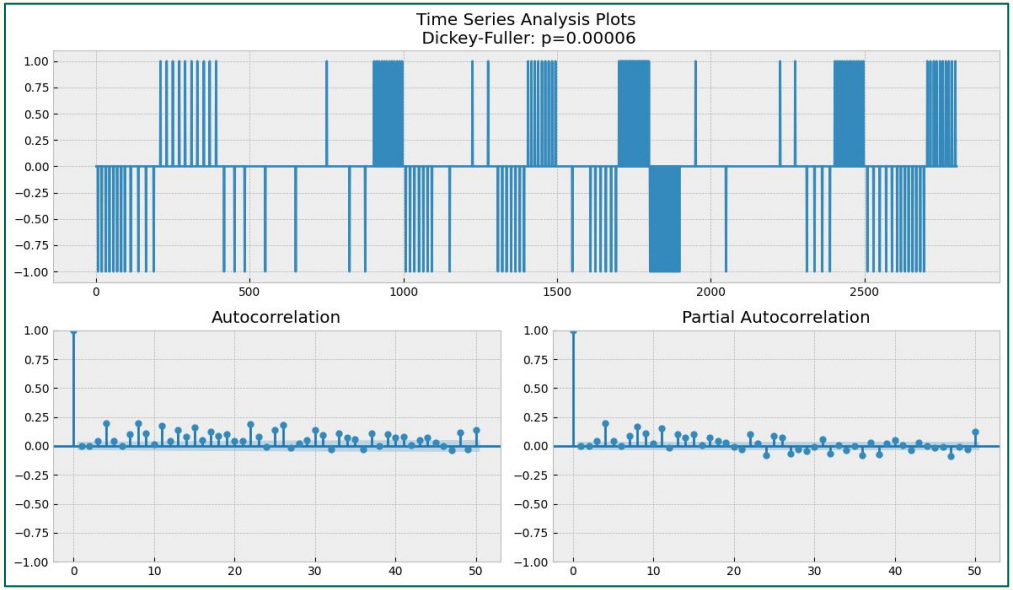


Exponential smoothing uses a similar logic to moving average, but this time, a different decreasing weight is assigned to each observations. We applied different alphas - 0.05 and 0.3.



Double exponential smoothing is used when there is a trend in the time series. It is also called Holt-Winters seasonal method. We apply alphas - 0.9 and 0.3 and betas - 0.9 and 0.3.

Analysis of Time Series Data



We conducted Dickey-Fuller test to check if the data is stationary and a p-value <0.05 confirms that it is. We also checked the Autocorrelation and Partial-Autocorrelation graphs to figure out the value of p and q respectively.

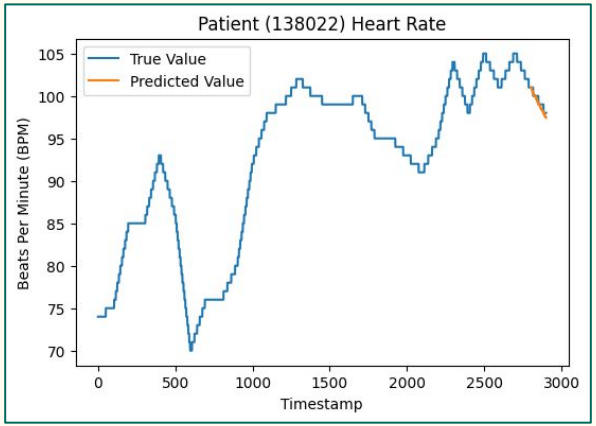
ARIMA

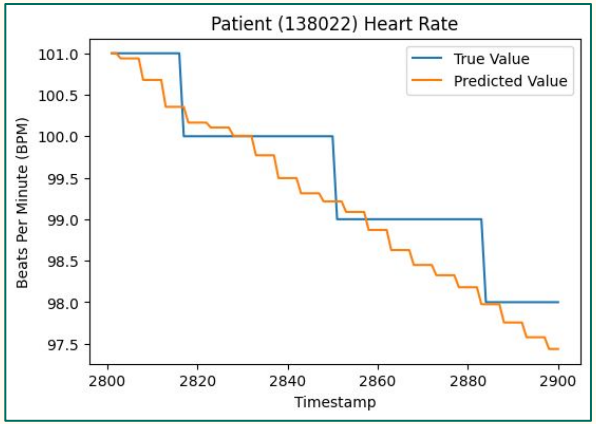
- ARIMA is an acronym that stands for **AutoRegressive Integrated Moving Average**. The key aspects of the model are:
  - *AR*: Autoregression. A model that uses the dependent relationship between an observation and some number of lagged observations.
  - *I*: Integrated. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
  - *MA*: Moving Average. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.
- ARIMA models are a general class of models used for forecasting time series data. ARIMA models are generally denoted as ARIMA (p,d,q) where p is the order of autoregressive model, d is the degree of differencing, and q is the order of moving-average model.

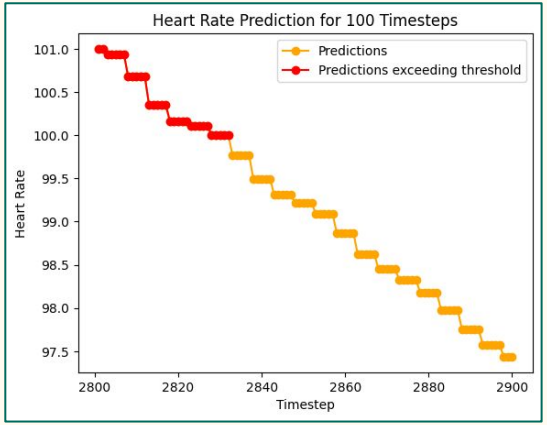
ARIMA Scenario 1: Easy Case - Heart Rate

- For scenario 1, the model is able to capture the inherent trend in the dataset for almost all the patients.
- The graphs show that not only the trends were followed, but the model was also able to predict values very close to the actual heart rate values.
- The graph below shows predicted values that exceed the normal heart rate range of 60-100 in red.

Patient ID: 138022



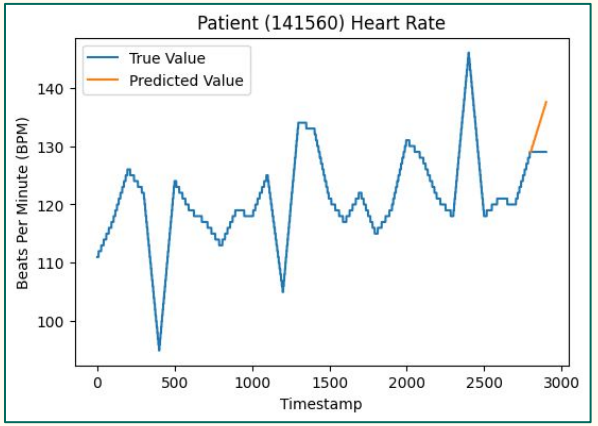


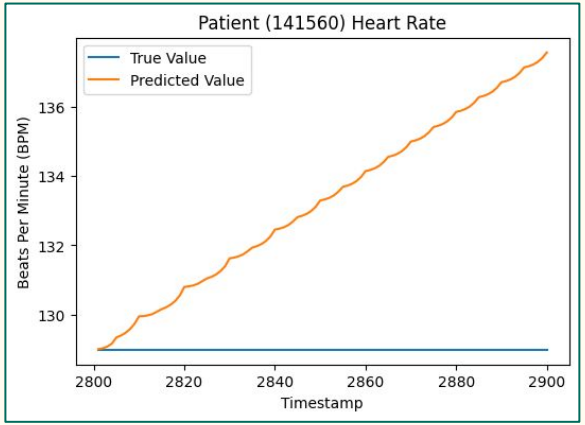


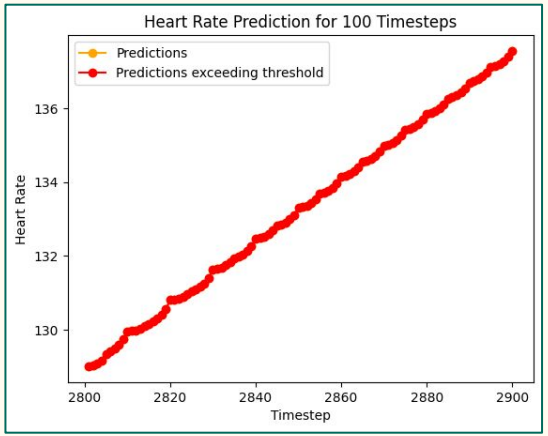
ARIMA Scenario 2: Hard Case - Heart Rate

- In this case, the model's values are not correct at all, which means that the time series is not showing any pattern i.e. new values are not moving averages of previous values
- The graph below shows all predicted values exceed the normal heart rate range of 60-100 in red.

Patient ID: 141560



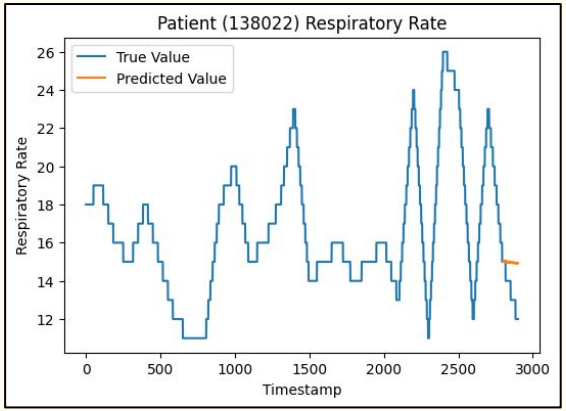


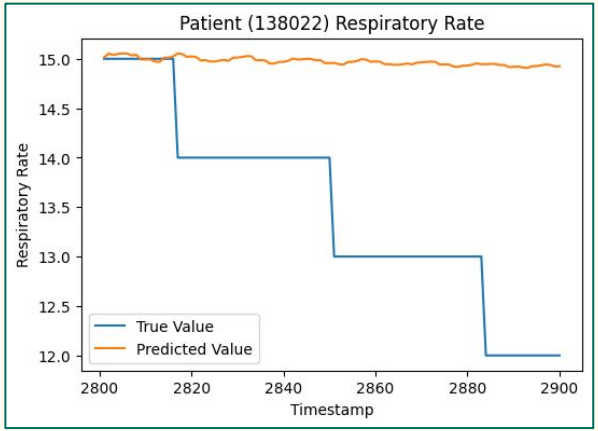


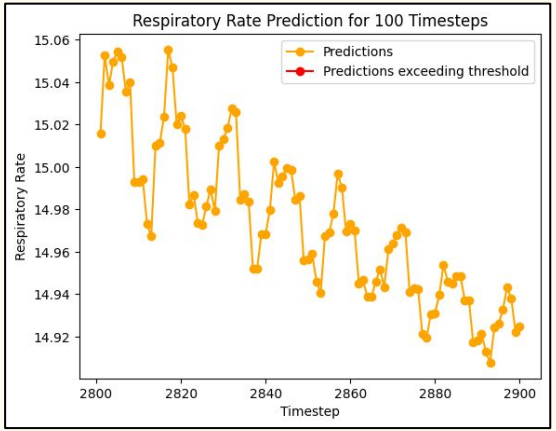
ARIMA Scenario 3: Easy Case - Respiratory Rate

- For scenario 1, we can see that ARIMA model is not able to forecast the values for next 100 timesteps.
- The graphs show that not only the trends were followed, but the model was also able to predict values close to the actual respiratory rate values.
- The graph below shows predicted values that exceed the normal heart rate range of 12-20 in red.

Patient ID: 138022

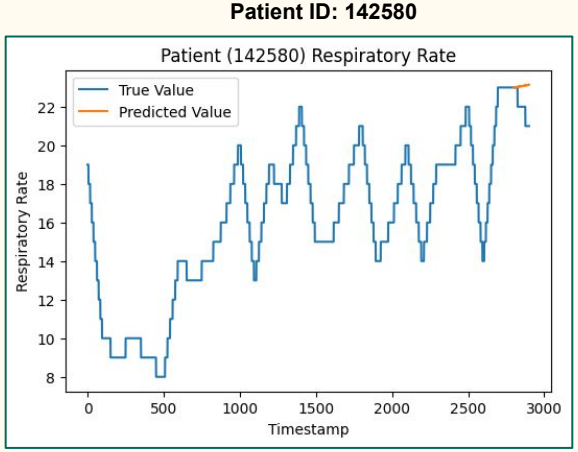


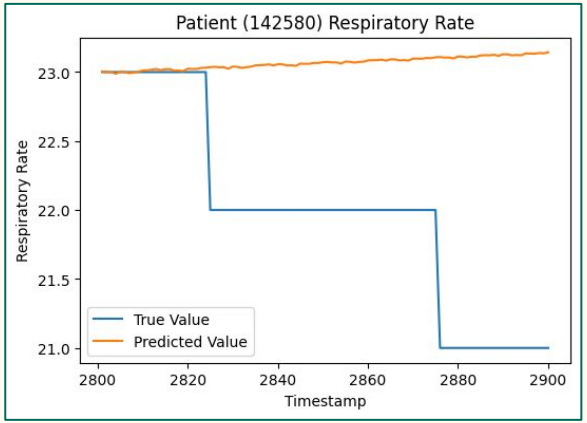


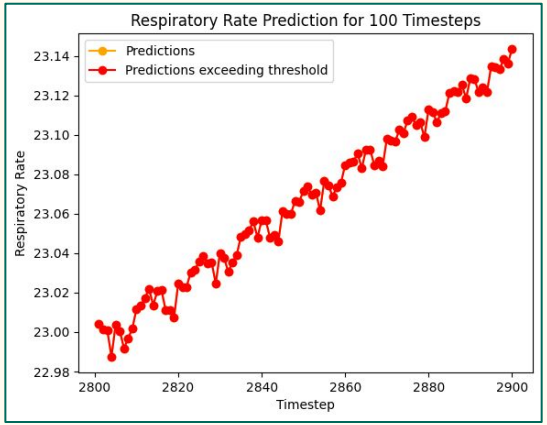


ARIMA Scenario 4: Hard Case - Respiratory Rate

- For scenario 2, the model's predictions are not good as the MAE is around 5.29.
- The graph below shows predicted values that exceed the normal respiratory rate range of 12-20 in red.



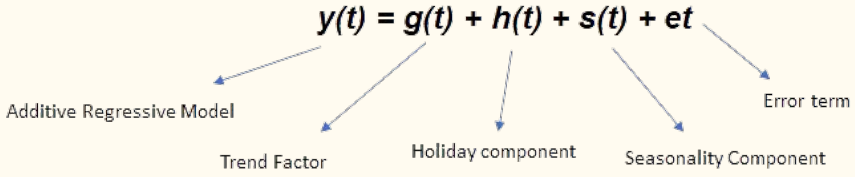




ARIMA - Analysis of the Results

- The model tends is not able to follow the trends. This is due to the limitation that the model is only taking moving averages and seasonality with the the assumption that the time series is stationary, which is not necessarily always true.
- ARIMA models assume a linear relationship in the time series data. If the data has nonlinear patterns, ARIMA may not capture these effectively. The model works best when the data is linear.
- As we can see that the data is only available for 4000 time steps, ARIMA models generally require a sufficiently long time series to accurately capture and forecast the data's patterns. Thus it can be one of the reasons for poor performance.
- Due to computational limitations, we failed to perform a grid search for the hyperparameter tuning which led to choosing the wrong order (p, d, q) for the ARIMA model that might have lead to poor performance.

PROPHET



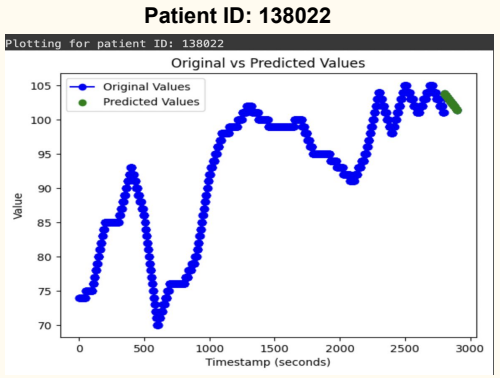
- Developed by Facebook for forecasting univariate time series data, especially effective for datasets with ***strong seasonal patterns.***
- The model adds up different factors like overall trend, seasonal patterns, and special events to make predictions.

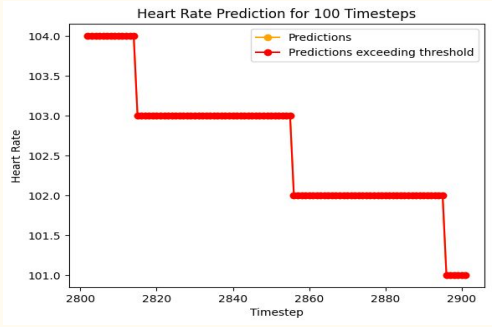
Use of Prophet in Health Monitoring:

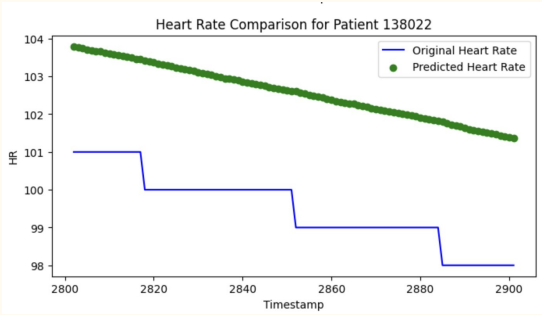
- ***Identifying long-term trends*** in vital signs (e.g., blood pressure, heart rate) for chronic disease management.
- Spotting ***unusual patterns*** or sudden changes in vital statistics that may indicate health issues.
- Forecasting future vital sign values to provide early warnings for potential health risks.
- Tailoring patient monitoring and care plans based on individualized trends and predictions.
- Assisting in planning and resource allocation in healthcare settings by predicting future monitoring needs.

Prophet Scenario 1: Easy case - Heart Rate

- For scenario 1, the model is able to capture the inherent trend in the dataset for almost all the patients but with compromised precision.
- The graphs show that although the trends were followed, the model was not able to predict values very close to the actual heart rate values.
- The model produces flat predictions for some time intervals. This could be due to the model's trend component, which may have inferred a level trend (no change) during these intervals based on the historical data patterns.
- The graph on the right shows predicted values that exceed the normal heart rate range of 60-100 in red.

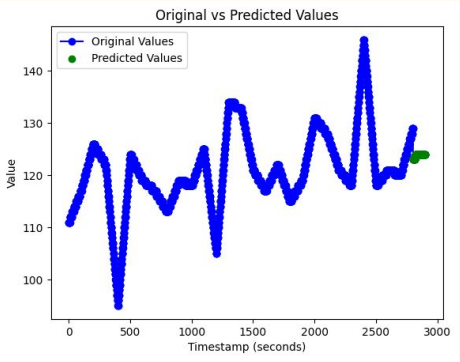




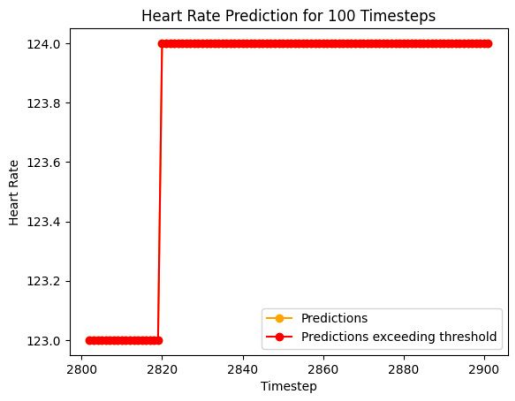


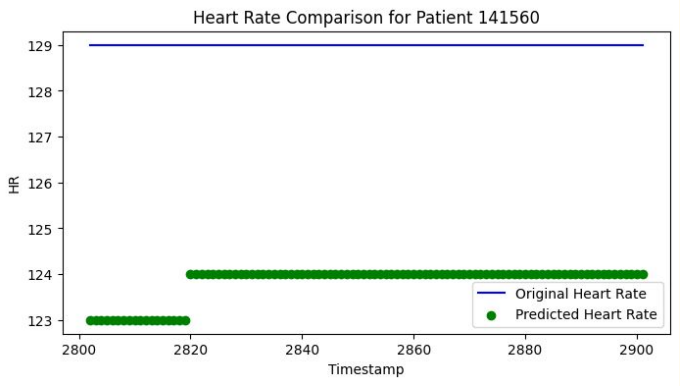
Prophet Scenario 2: Hard Case - Heart Rate

Patient ID: 141560



- In this case, the model once again sort of follows the trend but the predicted values are significant distant from the actual values.
- The graphs shows uncertainty with respect to the model following the trend making future predictions relatively unreliable.
- The graph below shows predicted values that exceed the normal heart rate range of 60-100 in red.

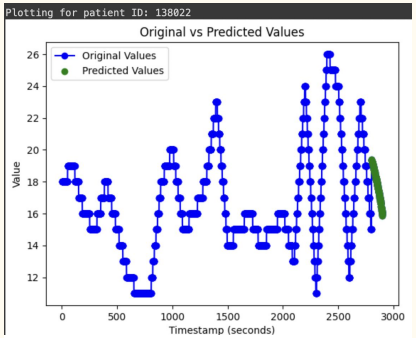


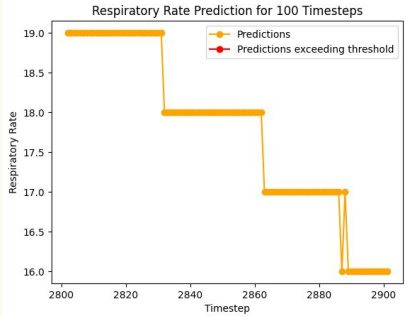


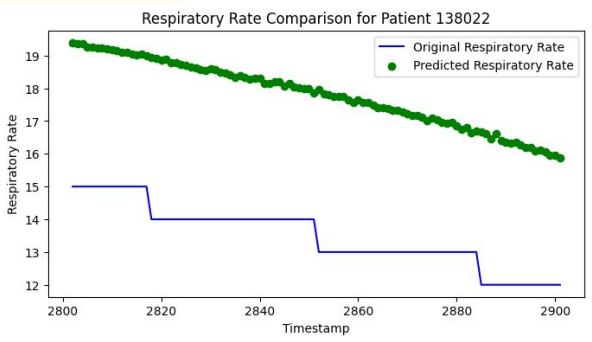
Prophet Scenario 3: Easy Case - Respiratory Rate

- For scenario 1, the model is able to capture a high level trend in the dataset for almost all the patients.
- The graphs show that the trends were followed to a certain extent with a few deviations.
- The model was not able to predict values very close to the actual respiratory rate values.
- The graph alongside shows that the predicted values do not seem to exceed the normal respiratory rate range of 12-20 in red.

Patient ID: 138022

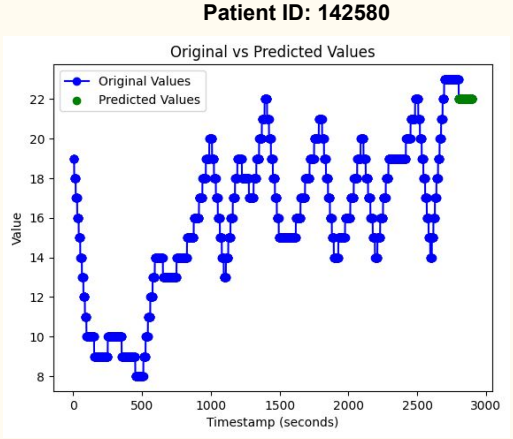


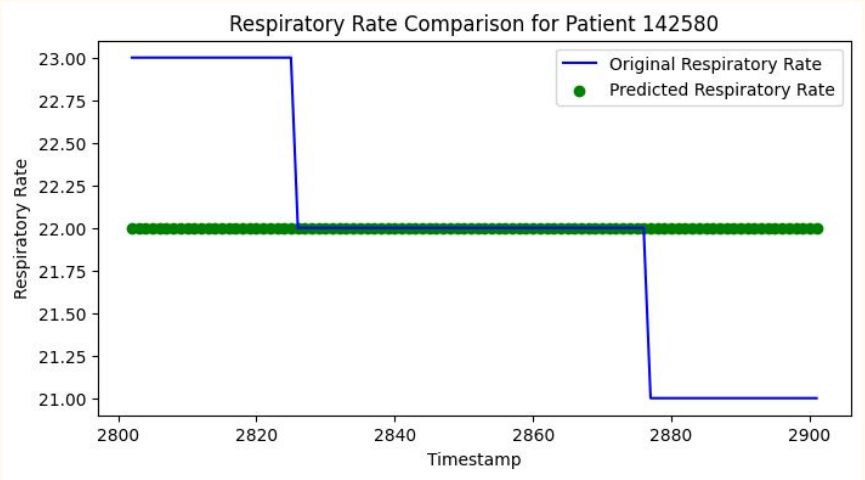


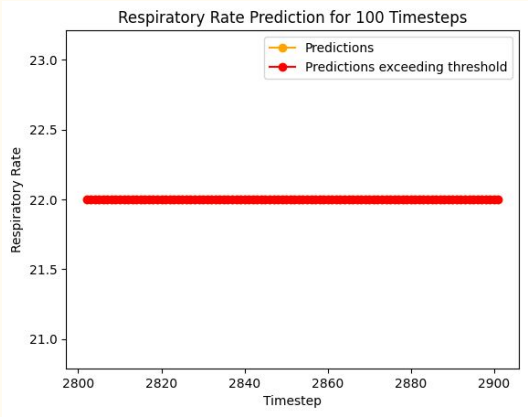


Prophet Scenario 4: Hard Case - Respiratory Rate

- For scenario 2, the model's predictions show no variation.
- The graphs show that neither the trend is followed and nor are the predicted values a match for the actual values, indicating its poor credibility.
- The graph below shows predicted values that exceed the normal respiratory rate range of 12-20 in red.





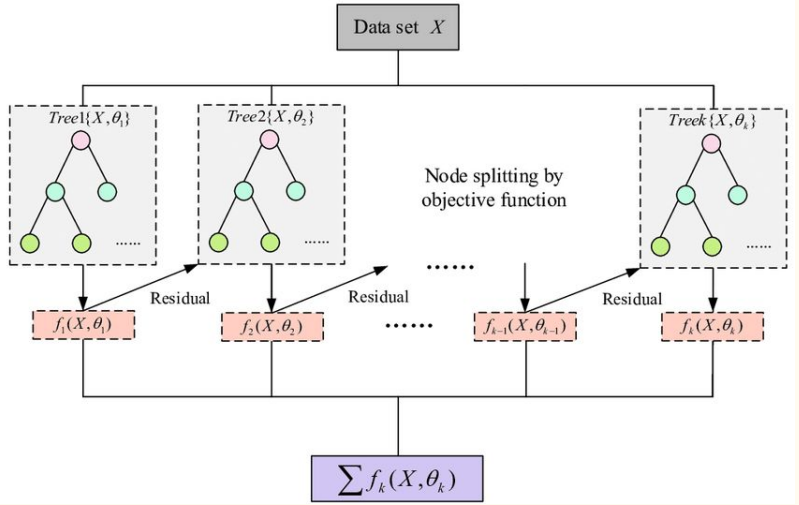


Prophet - Analysis of the Results

- The model tends to follow the trends but with deviations. These deviations could be due to the model capturing noise or anomalies within the data or could be a reflection of the model's sensitivity to changes.
- The graphs show smooth lines. The model's capacity to handle outliers and missing data can also lead to smooth predictions explaining why the predicted values sometimes do not capture the full extent of variability shown by the actual values
- If the data has any intrinsic seasonality (daily, weekly, etc.), Prophet would automatically try to model this. However, if the seasonality is not strong or is overpowered by the trend, the resulting predictions may appear as a smooth trend without capturing short-term fluctuations.
- Prophet is sensitive to changepoints in the data—times when the heart rate trend significantly changes. The model likely identified changepoints in the historical data, which informed the flat prediction segments as periods where the heart rate stabilizes before another shift.

XGBoost

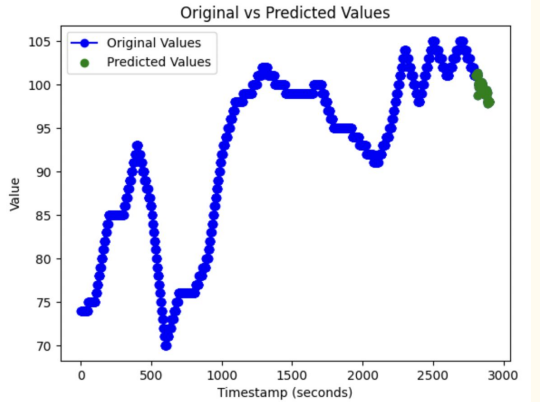
- ***eXtreme Gradient Boosting*** adds new trees by continuously splitting features.
- Adding a tree each time is actually learning a new function to fit the residual of the last prediction.
- The corresponding scores of each tree are added up to obtain the recognition prediction value of the sample.
- XGBoost is designed for ***speed and efficiency***, making it suitable for large datasets.
- It handles missing data, providing ***flexibility*** to adapt to different time series characteristics.
- It is ***robust to outliers***, making it effective for identifying key factors in time series data.



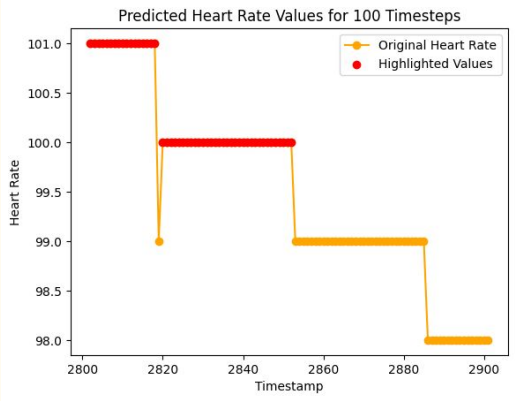
XGBoost flow chart

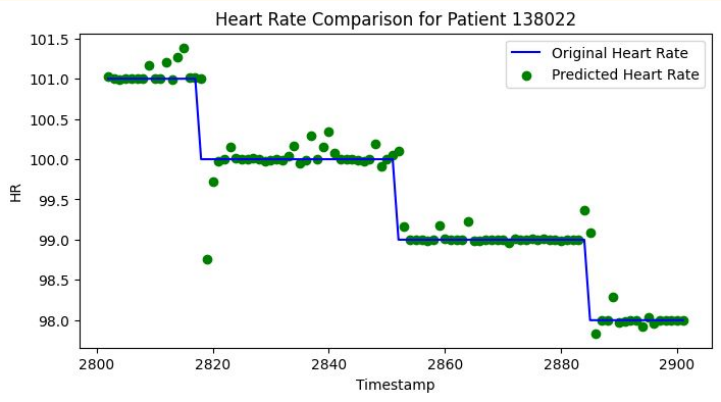
XGBoost Scenario 1: Easy case - Heart Rate

Patient ID: 138022



- For scenario 1, the model is able to capture the inherent trend in the dataset for almost all the patients.
- The graphs show that not only the trends were followed, but the model was also able to predict values very close to the actual heart rate values.
- The graph below shows predicted values that exceed the normal heart rate range of 60-100 in red.

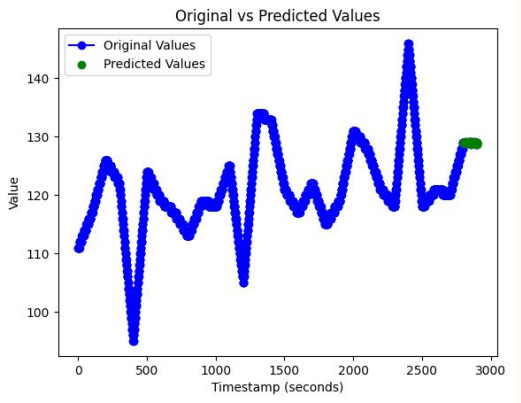


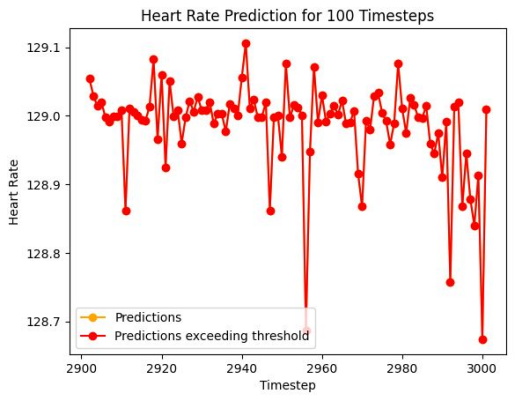


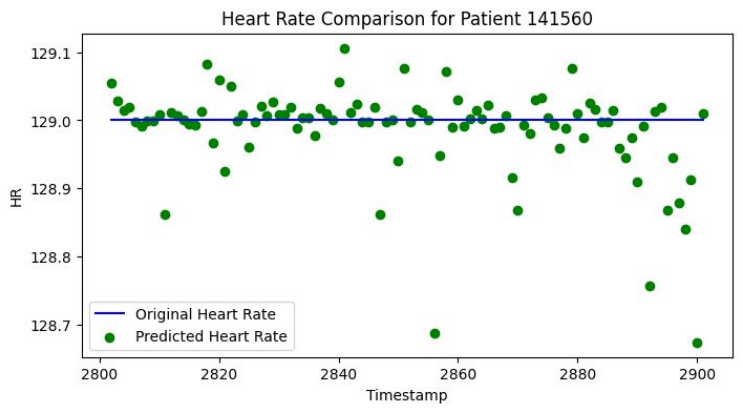
XGBoost Scenario 2: Hard Case - Heart Rate

Patient ID: 141560

- In this case, the model's predictions are not very certain. The predicted values seem to vary a lot.
- The graphs show that although the trend is somewhat followed for the 100 timesteps, it could fail in the future.
- The graph below shows predicted values that exceed the normal heart rate range of 60-100 in red.



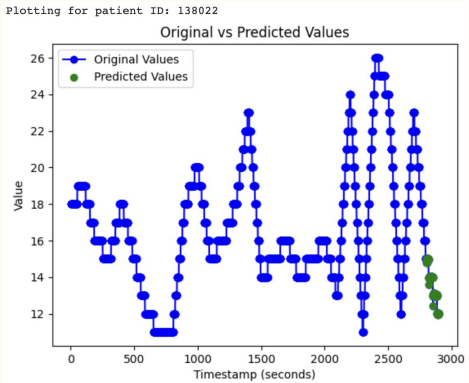


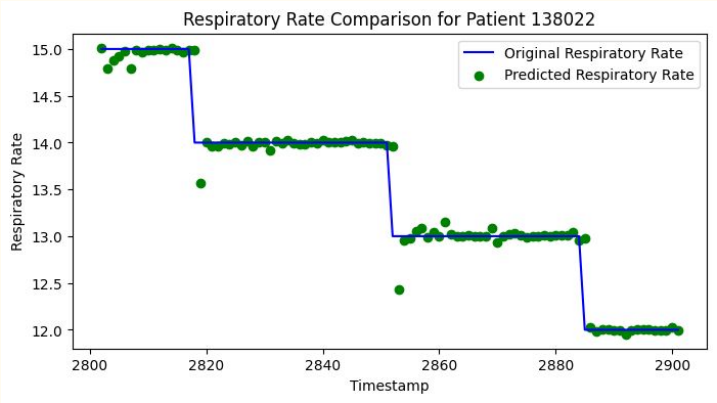


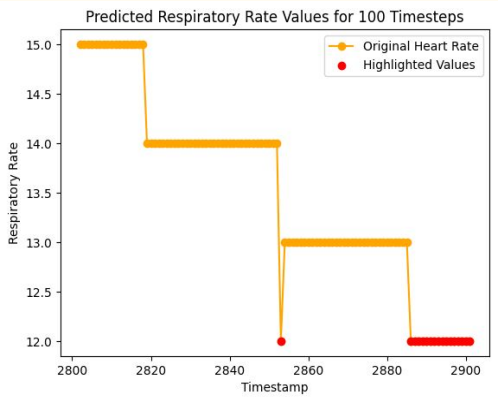
XGBoost Scenario 3: Easy Case - Respiratory Rate

- For scenario 1, the model is able to capture the inherent trend in the dataset for almost all the patients.
- The graphs show that not only the trends were followed, but the model was also able to predict values very close to the actual respiratory rate values.
- The graph below shows predicted values that exceed the normal respiratory rate range of 12-20 in red.

Patient ID: 138022

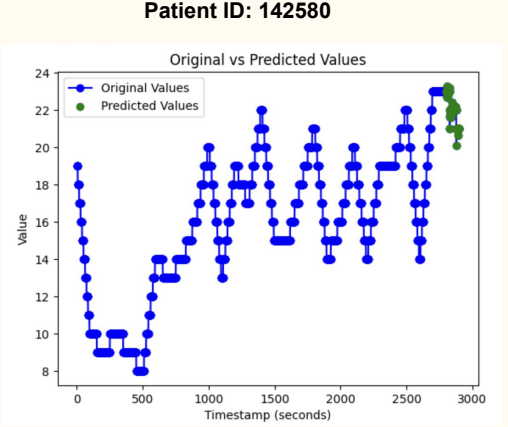


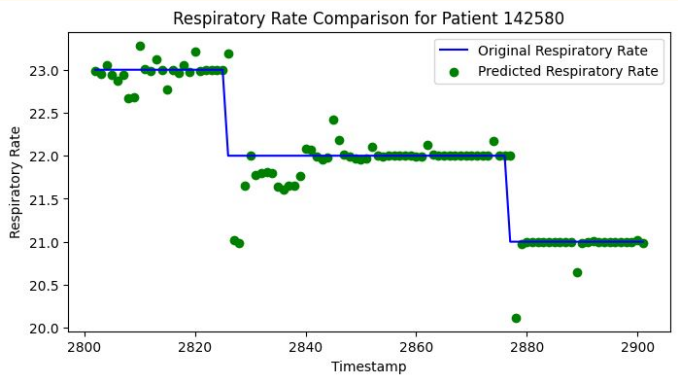


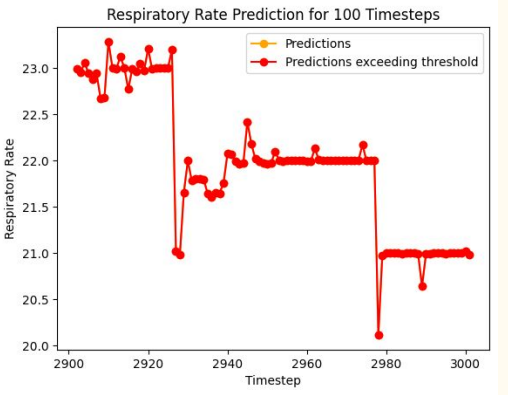


XGBoost Scenario 4: Hard Case - Respiratory Rate

- For scenario 2, the model's predictions are not very certain. The predicted values seem to vary a lot.
- The graphs show that although the trend is somewhat followed for the 100 timesteps, it could fail in the future.
- The graph below shows predicted values that exceed the normal respiratory rate range of 12-20 in red.







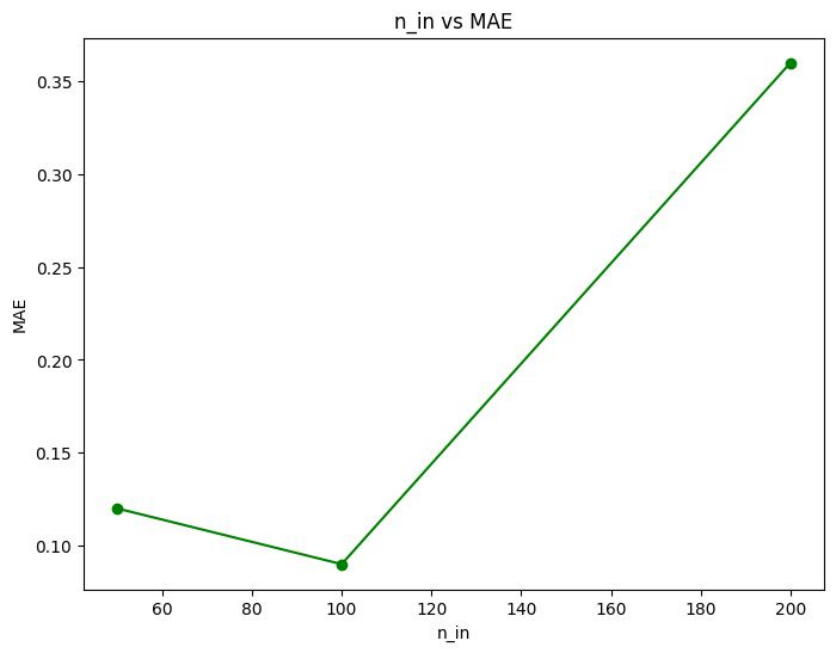
XGBoost - Analysis of the Results

- Reasons for XGBoost's Success in the Easy Case:
  - XGBoost excels in capturing ***nonlinear trends*** in medical datasets, leveraging its ability to model complex relationships through decision trees.
  - XGBoost's ensemble structure enables effective handling of ***temporal dependencies***, allowing it to accurately predict heart rate and respiratory rate trends over time.
- Challenges in the Hard Case:
  - XGBoost struggles in the hard case where underlying heart rate and respiratory rate patterns are complex and less consistent, leading to ***increased variability*** in predictions.
  - The model faces difficulty in ***generalizing learned patterns*** to future predictions, particularly when trends become less predictable, resulting in less certain and more variable forecasts.

Hyperparameter tuning

The main hyperparameters tuned in the implementation of ML models include:

1. **n_estimators**: The number of boosting rounds or trees to be built. Set to 1000 for XGBoost
2. **n_in**: represents the number of past time steps that are used as input features for the model. A larger n_in allows the model to capture longer-term dependencies in the data but may also increase the dimensionality of the input.
3. **n_out** : represents the number of future time steps to be predicted. A larger n_out implies that the model is making predictions further into the future.



As n_in was increased, so did the **Mean Average Error** of the predictions. A similar trend was followed for all the hyperparameters

Analysis of the ML models

- Comparing the three predictive models, XGBoost seems to perform the best.
- Both Arima and Prophet do not work well with non-linear and highly varying data.
- XGBoost does a better job of filling in missing values and identifying unique patterns.
- This ensemble approach of using multiple predictive models ensures overall better results as compared to using only one of these models.
- However, as the size of the dataset increases to 10 patients, the MAE value increases.

Avg. across 5 patients

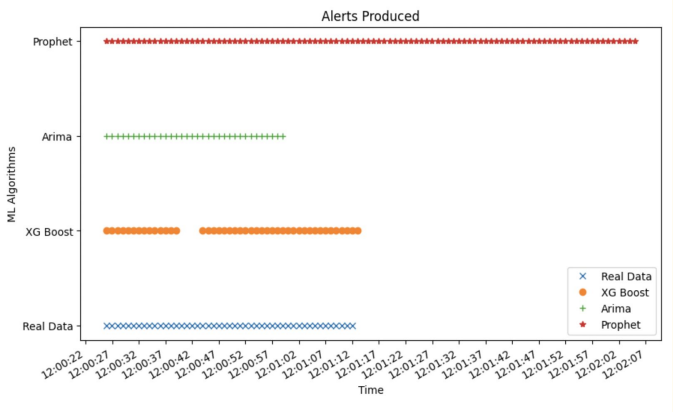
Model	Heart Rate MAE	Respiratory Rate MAE
Arima	0.35	1.48
Prophet	2.37	2.38
XGBoost	0.07	0.05

Avg. across 10 patients

Model	Heart Rate MAE	Respiratory Rate MAE
Arima	0.35	1.48
Prophet	3.88	1.12
XGBoost	0.09	0.06

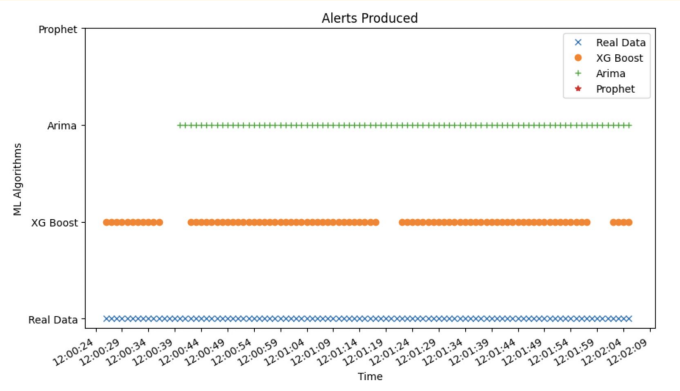
Comparing Alerts for Heart Rate

Scenario 1



Model	Arima	Prophet	XGBoost
Accuracy	0.87	0.47	0.95
Precision	1.0	0.47	0.977
Recall	0.72	1.0	0.914
F1-Score	0.839	0.639	0.945

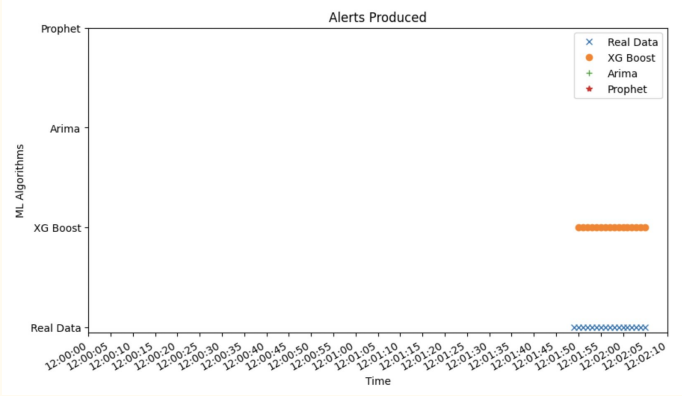
Scenario 2



Model	Arima	Prophet	XGBoost
Accuracy	0.86	0	0.87
Precision	1	N/A	1
Recall	0.86	0	0.87
F1-Score	0.92	0	0.93

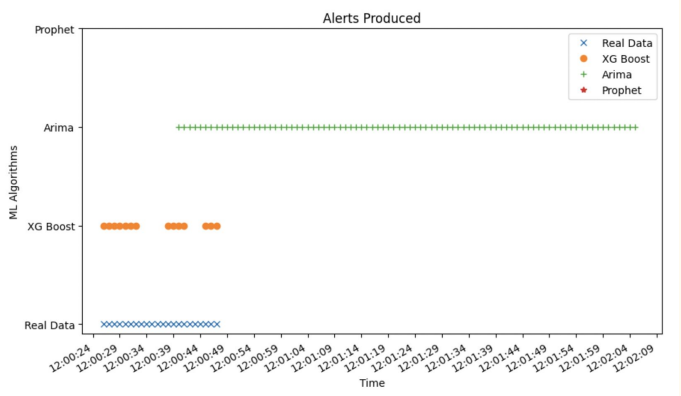
Comparing Alerts for Respiratory Rate

Scenario 3



Model	Arima	Prophet	XGBoost
Accuracy	0.83	0.83	0.99
Precision	N/A	N/A	1
Recall	0	0	0.94
F1-Score	0	0	0.97

Scenario 4



Model	Arima	Prophet	XGBoost
Accuracy	0.08	0.78	0.92
Precision	0.09	N/A	1
Recall	0.36	0	0.64
F1-Score	0.14	0	0.78

Analysis of Predictive vs Real Time Alerts

- In healthcare, **false negatives** can be very risky, as they might mean missing out on diagnosing a condition. Hence, a **high recall** is important. In the previous experiments, we saw that **XGBoost** generally provided this, except in one evaluation (0.64), but this still greater than other models.

- A **high precision** is also critical to **avoid** unnecessary treatments or anxiety due to false positives. Both **ARIMA** and **XGBoost** excel in this, but the consistency of XGBoost makes it more reliable.

- The variability in **Prophet's** performance suggests it might be less reliable for this particular application or that it requires further tuning and validation.

- **F1-Score** provides a balance between precision and recall, and **XGBoost** consistently shows the highest F1-Scores, suggesting it might be the most reliable model for deployment in healthcare scenarios.

Resources

- Dataset: https://physionet.org/content/challenge-2012/1.0.0/
- XGBoost: https://machinelearningmastery.com/xgboost-for-time-series-forecasting/
- Prophet: https://machinelearningmastery.com/time-series-forecasting-with-prophet-in-python/
- Arima: https://towardsdatascience.com/the-complete-guide-to-time-series-analysis-and-forecasting-70d476bfe775
- Apache Flink: https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/try-flink/datastream/