

Project 1

Blood Glucose (BG) Prediction and Control in Diabetes

Yonsei University

김호연, 송성모, 최예찬

Index

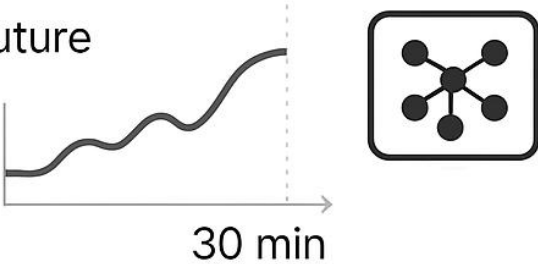
1. Problem formulation.
2. Data and methods.
3. Preliminary results.
4. Future plan and the expected outcomes.

Problem Definition

Blood Glucose Prediction

Goal: Predict blood glucose levels 30 minutes into future

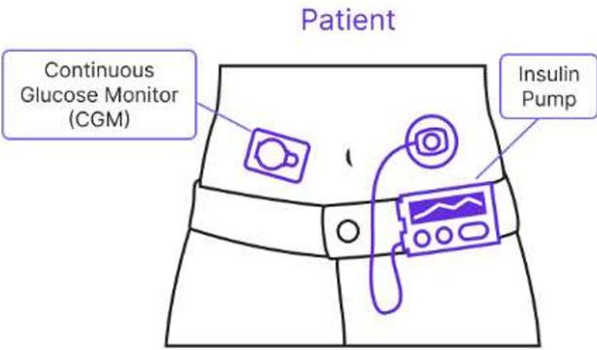
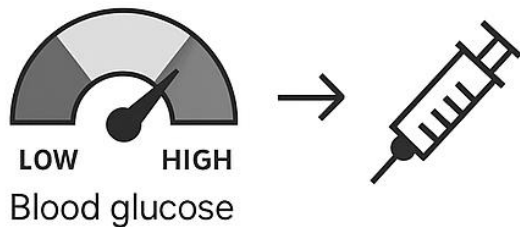
Requires trend analysis and predictive modeling
(e.g., LSTM, GRU, regression models)



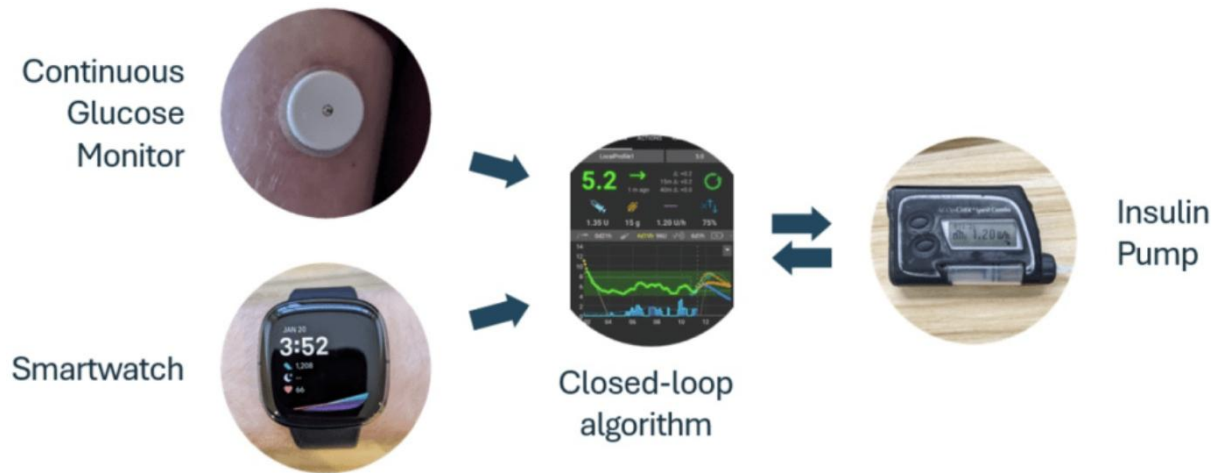
Insulin Dose Control

Goal: Determine the optimal insulin dose to maintain glucose within a normal range

Control policy is designed based on predicted glucose
and current physiological status
(e.g., reinforcement learning, optimal control theory)



2. Data and methods: Data Description.



Name	Description	Measurement Method
bg-X:XX	Blood glucose level measured	Measured via CGM device
insulin-X:XX	Insulin dose administered	Measured via insulin pump
carbs-X:XX	Amount of carbohydrate consumed	Self-reported
hr-X:XX	Average heart rate measured	Measured via wearable devices
steps-X:XX	Number of steps taken	Measured via wearable devices
cals-X:XX	Calories burned	Estimated via wearable devices
activity-X:XX	Type of physical activity performed	Self-reported via wearable devices

Data Collection Summary

- Participants: 24 people with Type 1 Diabetes (T1D)
- Duration: 6 months
- Devices: Participants used smartwatches (provided or personal) and T1D devices

Collected Data Sources:

- Real-time glucose levels from Continuous Glucose Monitors (CGM)
- User-entered carbohydrate intake
- Smartwatch data (heart rate, step count, calories burned, activity types)
- Insulin dosage from insulin pumps

Data Structure:

- For each event (e.g., glucose reading, insulin injection, meal), data from the preceding ~5 minutes is collected (X hours XX minute prior to the event)

2. Data and methods: Data Distribution.

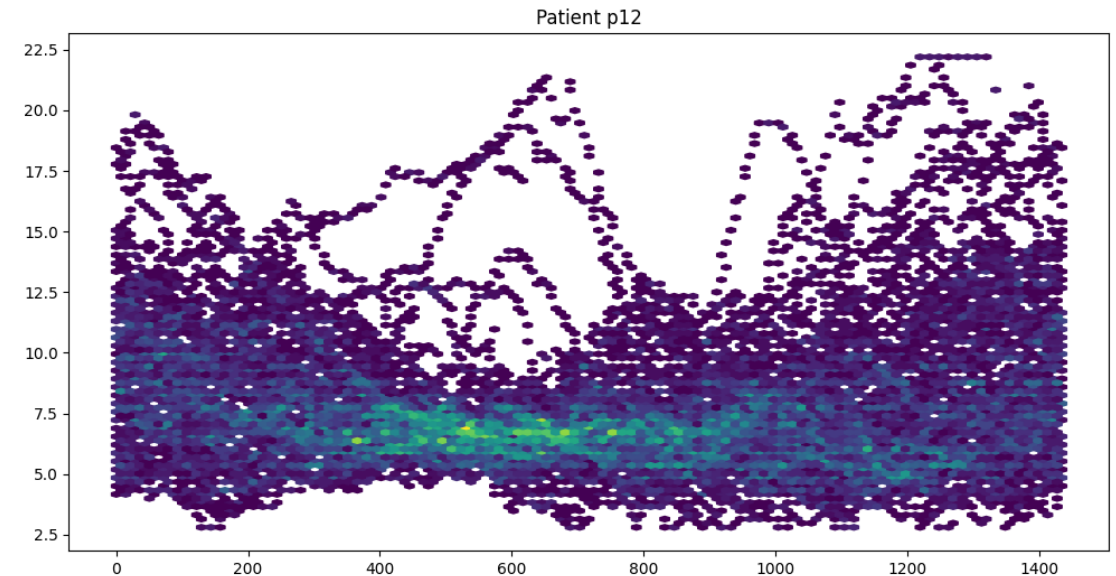
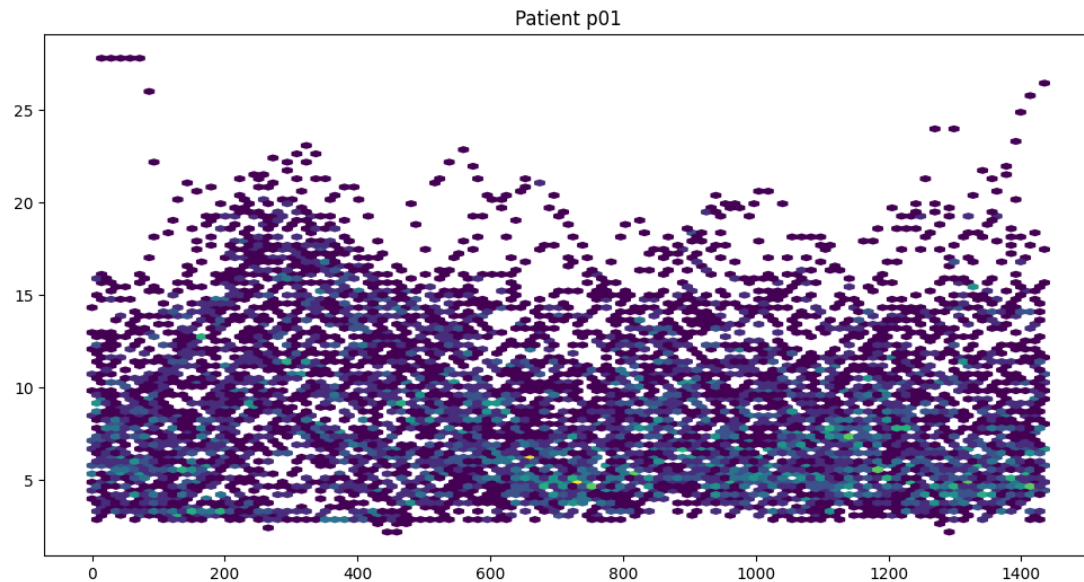
1) Hexbin Plot

Patient p01 shows **highly variable and irregular glucose patterns**.

→ It is hard to extract clear rules from past data, so a **Transformer-based time series model** is more appropriate.
(Capable of capturing complex sequential dependencies and abrupt changes)

Patient p12 shows **regular and stable glucose patterns**.

→ Simple statistical patterns are sufficient, making a **Tree-based regression model** (e.g., LightGBM) a strong candidate.



2. Data and methods: Data Distribution.

2) KDE Plot

Gap=1 (5-minute interval) is more sensitive to real-time fluctuations.

→ It is well-suited for detecting rapid blood glucose changes caused by **meals, exercise, or insulin injections**.

Gap=3 (15-minute interval) reflects changes **after they have already occurred**,

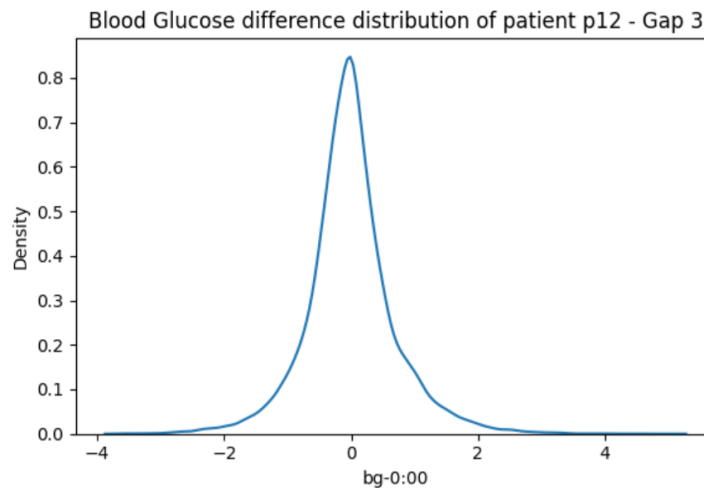
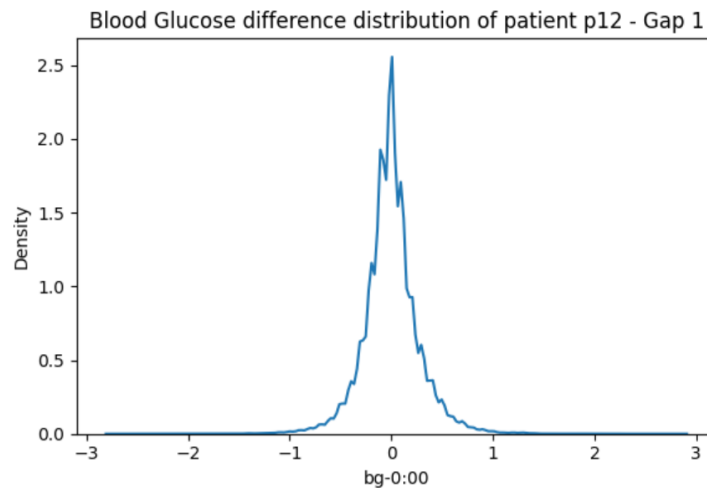
→ making it harder to capture **fine-grained, immediate fluctuations** due to smoothing over time.

Experimental Insight (based on Patient p12)

The **KDE plot for Gap=1** shows a **sharp and narrow peak near 0**, indicating a high frequency of small, stable changes.

In contrast, the **Gap=3 KDE plot** is **more spread out**, diluting fine variations.

→ This suggests that Gap=3 may obscure subtle patterns in glucose fluctuations.



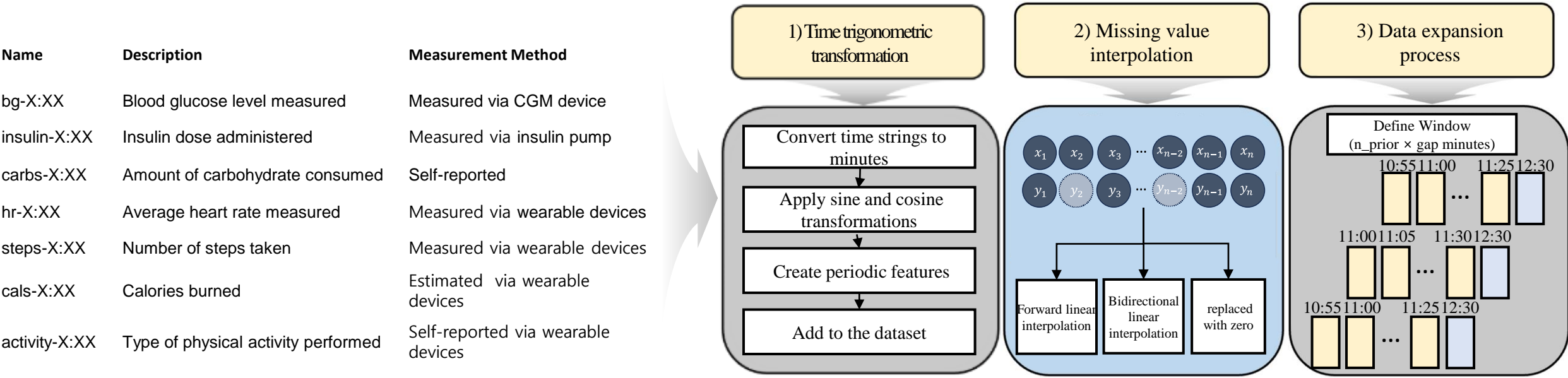
2. Data and methods: Preprocess.

1) Motivation

- Generate a fixed-length feature vector for predicting blood glucose levels one hour ahead (bg+1:00)
- Transform and integrate diverse physiological signals into a machine learning-ready format

2) Overview

- The pipeline resolves sampling inconsistencies, imputes missing values, and engineers temporal features for predictive modeling



Use GAP=1,
n_prior=12 (1 hour)

2. Data and methods: Preprocess.

1) Data window expansion process - `data_expander(df, gap, n_prior, addition, data_source)`

- Define a prediction time and collect past data for a fixed time window ($n_prior \times \text{gap}$ minutes).
- Shift all time-series features (e.g., blood glucose, insulin) backward to simulate earlier time points.
- Assign the target value (e.g., $\text{bg}+1:00$) corresponding to the prediction horizon.
- Repeat the process with sliding time steps to generate multiple training samples per patient.

2) Time trigonometric transformation - `get_time_in_minutes_trig(series)`

- **Convert time strings to minutes:** Time values like "14:30" are converted into numeric values: $14 \times 60 + 30 = 870$ minutes.
 - **Apply sine and cosine transformations:** Time in minutes is transformed into two new features using trigonometric functions:
 - $\sin(2\pi \times \text{minutes} / 1440)$
 - $\cos(2\pi \times \text{minutes} / 1440)$
 - This captures the cyclical nature of time within a 24-hour period (1440 minutes).
- Incorporating time-of-day as cyclic features helps the model capture daily rhythms in blood glucose, ultimately improving predictive performance.

3) Missing value interpolation

- **Blood glucose (BG):** Forward linear interpolation with a limit of 3 consecutive missing values.
- **Insulin:** Bidirectional fill with a tolerance of 3 steps. Filled only if forward and backward values match; otherwise set to 0.
- **Carbohydrates:** All missing values are filled with 0 (no interpolation used).
- **Steps:** Bidirectional fill with a tolerance of 3 steps, similar to insulin.
- **Heart rate (HR):** Bidirectional linear interpolation with a limit of 3 consecutive missing values.
- **Calories (Cals):** Bidirectional linear interpolation with a limit of 3.
- **Activity:** All non-missing values are converted to 1; missing values are set to 0 (binary masking, no interpolation).

```
def expand(gap, n_prior, addition):  
    # Expands both train and test data  
    # Expanding train data won't generate much additional data, but expanding test data will  
    # The step list stores the iterations from the expansion. Given that it is a list, it is not a DataFrame  
    make_dir(Path(WORK_DIR))  
  
    train_df = pd.read_csv(Path(RAW_TRAIN_FILE), low_memory=False)  
    test_df = pd.read_csv(Path(RAW_TEST_FILE), low_memory=False)
```

```
# create trigonometric features from time  
def get_time_in_minutes_trig(series):  
    split_time = series.str.split(":", expand=True).astype(int)  
    time_in_minutes = split_time.iloc[:, 0] * 60 + split_time.iloc[:, 1]  
    time_in_minutes_sin = np.sin(2 * np.pi * time_in_minutes / 1440)  
    time_in_minutes_cos = np.cos(2 * np.pi * time_in_minutes / 1440)  
    time_trig_df = pd.concat([time_in_minutes_sin, time_in_minutes_cos], axis=1)  
    return time_trig_df
```

```
def blood_glucose_interpolation_stats(df, gap, n_prior, window=None):  
    X = pd.DataFrame(index=df.index)  
    df = df.interpolate(axis=1, limit=3, limit_direction='forward')
```

```
def insulin_interpolation_stats(df, gap, n_prior):  
    X = pd.DataFrame(index=df.index)  
    df = fillna_horizontal_with_tolerance(df, tolerance=3)
```

```
def carbs_interpolation_stats(df, gap, n_prior):  
    X = pd.DataFrame(index=df.index)  
    df = df.fillna(0)
```

```
def heart_rate_interpolation_stats(df, gap, n_prior):  
    X = pd.DataFrame(index=df.index)  
    df = df.interpolate(axis=1, limit=3, limit_direction='both')
```


2. Data and methods: Model.

1) Feature Selection

- To focus on short-term glucose dynamics, we excluded time-lagged features older than 30 minutes.
- Only features from 0 to 30 minutes before the prediction point were used as input variables.
- (e.g., bg-0:05, insulin-0:10, ..., carbs-0:30)

```
columns_to_remove = ([
    + [f'bg_{lag}_lag' for lag in [35,40,45,50,55,60]]
    + [f'bg_{lag}_diff' for lag in [30,35,40,45,50,55]]
])
```

2) Patient Encoding (p_num)

- The patient ID (p_num) cannot be used directly in the model.
- Instead, we computed statistical features from each patient's glucose values:
- Mean / Standard Deviation (std) / Skewness / Kurtosis
- These features help capture patient-specific glucose behavior in a generalized way.

```
def pipeline_transformer_creator(X, columns_to_remove=None, target_encoders=None):
    cols = X.columns.to_list()

    if columns_to_remove is not None:
        cols = [col for col in cols if col not in columns_to_remove]
    cols.remove('p_num')

    encoders = []
    if target_encoders is not None:
        if 'mean' in target_encoders:
            encoders.append(('mean', MeanEncoder(), ['p_num']))
        if 'std' in target_encoders:
            encoders.append(('std', StandardDeviationEncoder(), ['p_num']))
        if 'skew' in target_encoders:
            encoders.append(('skew', SkewnessEncoder(), ['p_num']))
        if 'kurt' in target_encoders:
            encoders.append(('kurt', KurtosisEncoder(), ['p_num']))

    pipeline_transformer = ColumnTransformer(
        transformers=encoders + [('col', 'passthrough', cols)],
        remainder='drop'
    )

    pipeline_transformer.set_output(transform="pandas")
    return pipeline_transformer
```

Motivation

- Limiting inputs to the most recent 30 minutes reduces irrelevant historical noise.
- Statistical encoding enables the model to learn personalized patterns without overfitting to individual IDs.

```
if regressor_type == 'lgbm':
    regressor = LGBMRegressor(**params)
    # other regressors can be added here
```

```
model = Pipeline(
    steps=[
        ('transform', transformer),
        ('regressor', regressor),
    ]
)

model.fit(X, y)
```

2. Data and methods: Model.

3) Model and Optimization

Model Selection

- Use LGBMRegressor, a fast and efficient gradient boosting model for tabular data.

Cross-Validation Strategy

- Apply TabularExpandingWindowCV,
- a time-aware cross-validation method that respects temporal order

Hyperparameter Tuning

- The following parameters were tuned to find the optimal model
 - num_leaves
 - min_data_in_leaf
 - min_sum_hessian_in_leaf
 - reg_alpha, reg_lambda
 - min_gain_to_split
 - subsample, colsample_bytree

Evaluation Metric

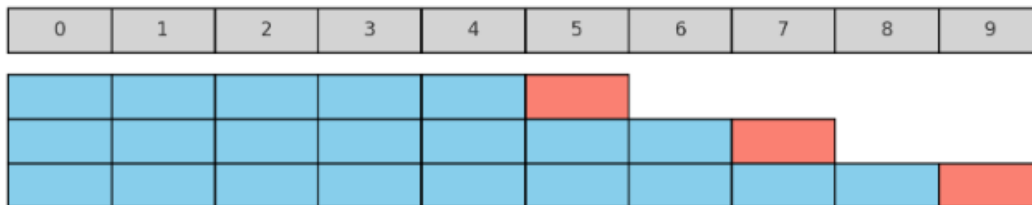
- Model performance was evaluated using Root Mean Squared Error (RMSE).

```
regressor.fit(  
    X_train,  
    y_train,  
    # sample_weight=train_weights,  
    eval_set=[(X_test, y_test)],  
    # eval_sample_weight=[(test_weights)],  
    eval_metric="rmse",  
    callbacks=[  
        # lgb.log_evaluation(  
        #     100  
        # ),  
        lgb.early_stopping(  
            stopping_rounds=250,  
            # min_delta=0.001,  
            verbose=False  
        ),  
        # LightGBMPruningCallback(trial, 'rmse')  
    ]  
)  
  
best_n_estimators = regressor.best_iteration_  
preds = regressor.predict(X_test)  
  
cv_scores[idx] = root_mean_squared_error(y_test, preds)  
cv_n_estimators[idx] = best_n_estimators
```

2. Data and methods: Cross Validation.

```
5 class TabularExpandingWindowSplitter(BaseCrossValidator):
6     def __init__(self, initial_window, step_list, initial_step, step_length, forecast_horizon):
7         self.initial_window = initial_window
8         self.step_list = np.array(step_list)
9         self.initial_step = initial_step
10        self.step_length = step_length
11        self.forecast_horizon = forecast_horizon
12
13    def split(self, X, y=None, groups=None):
14        # Define indices for the fixed portion
15        initial_window_indices = np.arange(self.initial_window)
16        remaining_indices = np.arange(len(initial_window_indices), len(X))
17        remaining_step_list = self.step_list[len(initial_window_indices): len(X)]
18
19        last_step = self.step_list.max()
20        n_splits = (((last_step - self.forecast_horizon) - self.initial_step) // self.step_length) + 1
21
22        for k in np.arange(n_splits):
23            last_step_window = self.initial_step + k * self.step_length
24            window_indices = (len(initial_window_indices) + np.arange(len(remaining_step_list)))[remaining_step_list <= last_step_window]
25            forecast_step = last_step_window + self.forecast_horizon
26            forecast_indices = (len(initial_window_indices) + np.arange(len(remaining_step_list)))[remaining_step_list == forecast_step]
27
28            train_indices = np.concatenate([initial_window_indices, window_indices])
29            test_indices = forecast_indices
30            yield train_indices, test_indices, forecast_step
31
32    def get_n_splits(self, X=None, y=None, groups=None):
33        last_step = self.step_list.max()
34        n_splits = (((last_step - self.forecast_horizon) - self.initial_step) // self.step_length) + 1
35        return n_splits
```

Expanding Window Cross-Validation (forecast_horizon = 1)



TabularExpandingWindowCV Class (Using BaseCrossvalidation)

This generator class splits time-series data into Train/Test sets, **gradually expanding the size of the training window** at each step. It outputs **sequential folds according to the specified step size**.

Key Function Description

`__init__` method:

Initializes parameters for expanding the training window.

`split` method:

Generates sequential Train/Test datasets using the initialized parameters

(e.g., initial window size, step size, forecast horizon).

Each call yields a new fold:

→ Train/Test Set 1, Train/Test Set 2, ...

Characteristics of the Validation Split Strategy

- Folds are created by **gradually increasing the proportion of training data**, considering the sequential nature of time-series data.

- This method **preserves the chronological order**, allowing the model to simulate learning from more data over time.

2. Data and methods: Cross Validation.

```
for idx, (train_idx, test_idx) in enumerate(cv.split(X)):
    print(f'running fold n° {idx + 1}')
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]
    # train_weights, test_weights = weights.iloc[train_idx], weights.iloc[test_idx]

    X_train = transformer.fit_transform(X_train, y_train)
    X_test = transformer.transform(X_test)

    regressor.fit(
        X_train,
        y_train,
        # sample_weight=train_weights,
        eval_set=[(X_test, y_test)],
        # eval_sample_weight=[(test_weights)],
        eval_metric="rmse",
        callbacks=[
            # lgb.log_evaluation(
            #     100
            # ),
            lgb.early_stopping(
                stopping_rounds=250,
                # min_delta=0.001,
                verbose=False
            ),
            # LightGBMPruningCallback(trial, 'rmse')
        ]
    )

    best_n_estimators = regressor.best_iteration_
    preds = regressor.predict(X_test)

    cv_scores[idx] = root_mean_squared_error(y_test, preds)
    cv_n_estimators[idx] = best_n_estimators

    print(f'RMSE: {cv_scores[idx]:.5f} at {best_n_estimators} estimators')
    sleep(1)

    # trial.report(cv_scores[idx], idx)
    # if trial.should_prune():
    #     raise TrialPruned()

trial.set_user_attr("fold_rmse", cv_scores)
trial.set_user_attr("best_n_estimators", cv_n_estimators)
return np.mean(cv_scores)
```



Cross-Validation Procedure Using TabularExpandingWindowCV

- Hyperparameter tuning is performed using the custom **TabularExpandingWindowCV splitter**.
- Each set of hyperparameters is treated as a separate **trial**.
- For each trial, the model is evaluated across all Train/Test folds generated by the splitter, **and RMSE values are computed**.
- The **average RMSE** across all folds within each trial is calculated.
- The trial with the **lowest average RMSE** is selected as the optimal hyperparameter set.

3. Preliminary results.

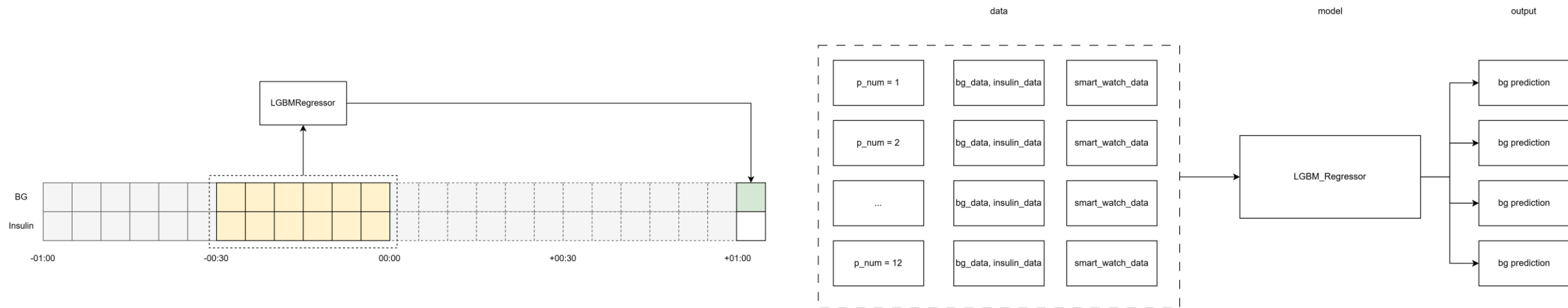
1) Model Performance

- The tuned **LGBMRegressor** showed improved prediction accuracy on the validation set.
- Cross-validation RMSE was significantly lower than the baseline mode

	lgbm_gap_1_prior_12_addition_0_submission_standard.csv	2.3938	2.3796
	Complete (after deadline) · 2h ago	Without smartwatch	
	lgbm_gap_1_prior_12_addition_0_submission_standard.csv	2.3615	2.3448
	Complete (after deadline) · 6h ago	With smartwatch	

2) Key Observations

- Features within the **last 30 minutes** were highly predictive of near-future glucose trends.
- Patient-level statistical encoding improved model generalization across individuals.



3. Preliminary results.

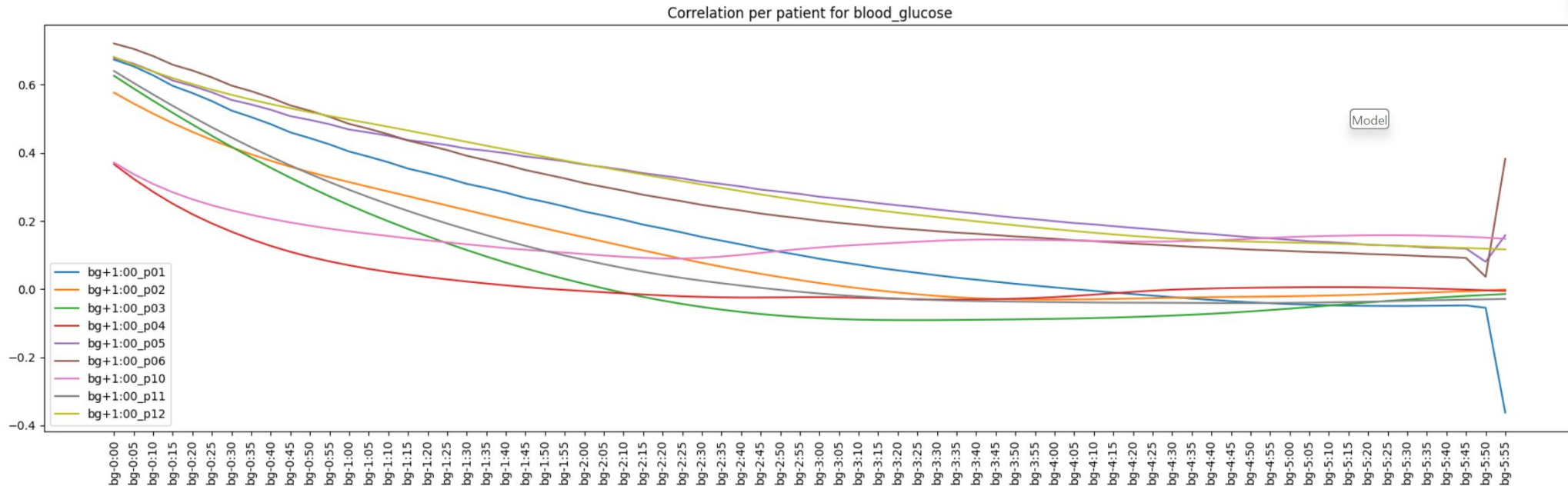
3) Mitigate

Across all patients, **recent blood glucose (BG) readings** show the **strongest correlation** with future BG (bg+1:00).

However, the **rate and pattern of correlation decay** vary by patient:

- Some patients (e.g., **p01, p05**) are sensitive to **short-term fluctuations**.
- Others (e.g., **p02, p11**) show stronger dependence on **long-term trends**.
- In some cases (e.g., **p06**), older data even has **negative correlation**, potentially **interfering with prediction**.

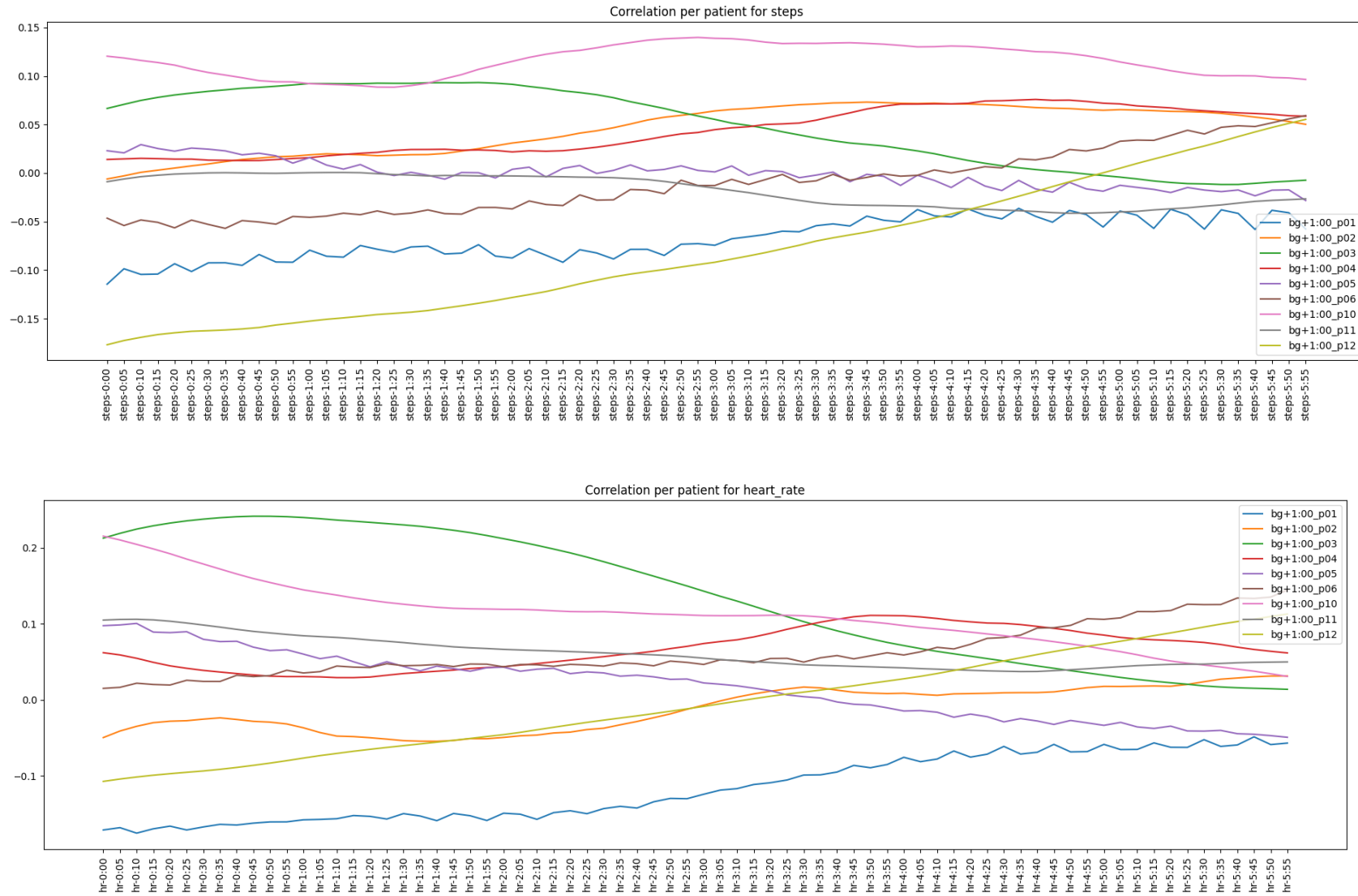
→ Transformer Can Mitigate Patient-Specific Correlation Differences



3. Preliminary results.

For the smartwatch data, it is more specific with patent.

→ Transformer Can Mitigate Patient-Specific Correlation Differences (as same as BG)



3. Preliminary results.

Insulin doses within the recent 0–1 hour window tend to show a **positive correlation** with future blood glucose.

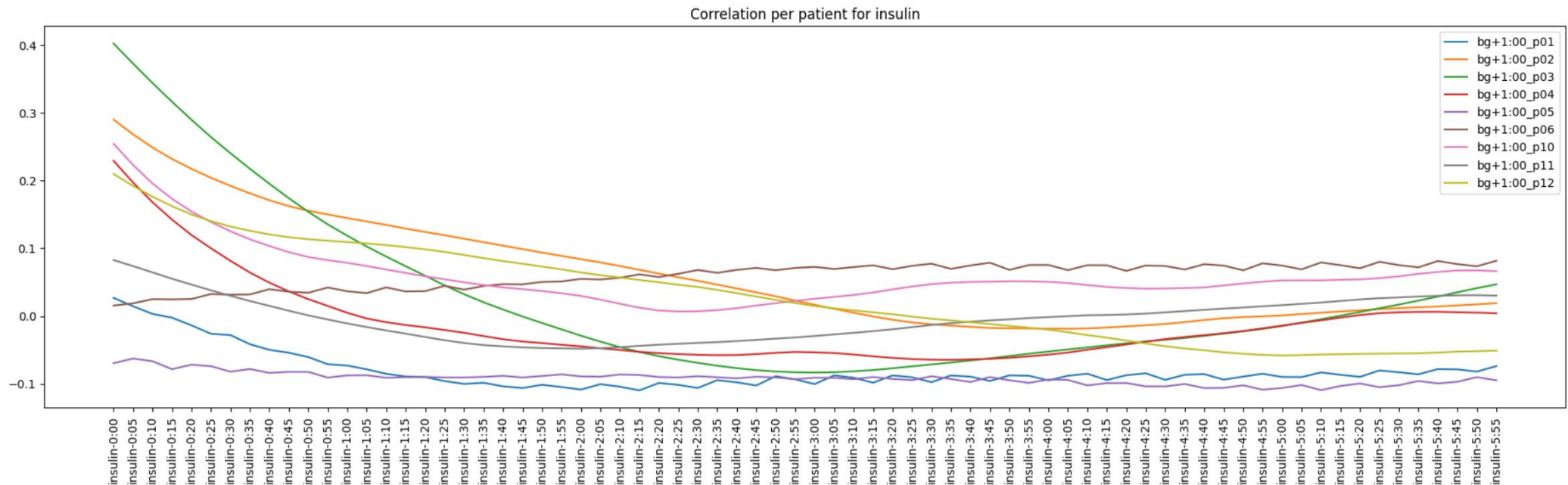
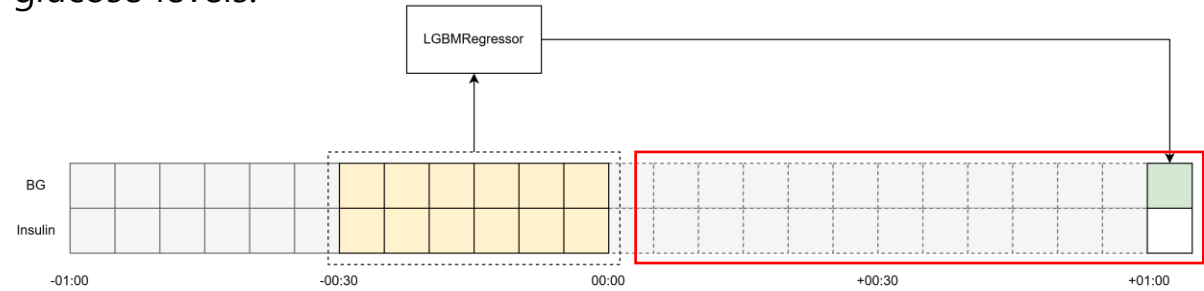
→ This occurs because insulin often hasn't taken full effect immediately after injection, resulting in glucose levels moving in the same direction as recent doses.

As more time passes, the correlation shifts to negative,

→ indicating that insulin has begun to take effect and is reducing blood glucose levels.

The **response time and sensitivity to insulin vary across patients:**

- **Fast response:** e.g., *p03*
- **Delayed response:** e.g., *p06*
- **Flat or minimal response:** e.g., *p10*



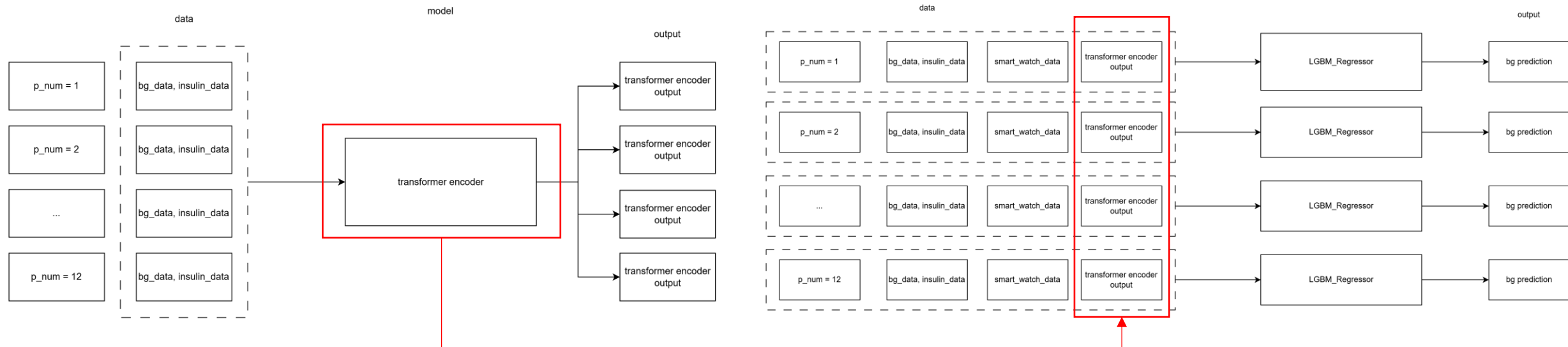
4. Future plan and the expected outcomes.

1) Planned Model Architecture

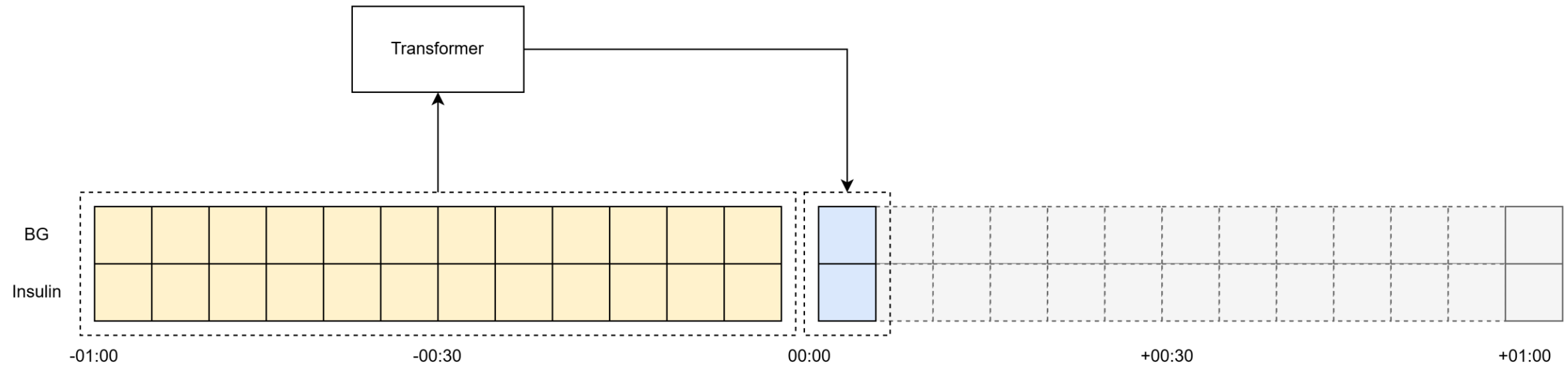
- Use a **Transformer-based model** to jointly predict:
 - Future blood glucose trends
 - Future insulin dosage needs
- The output of the Transformer will serve as an intermediate signal.

2) Glucose Control Module

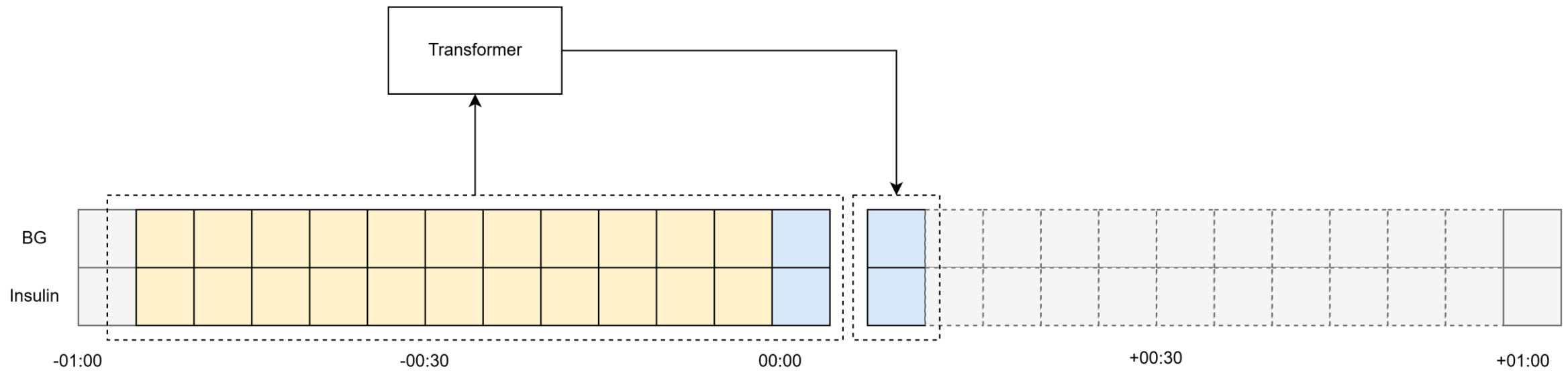
- A **tree-based regression model** will then take the Transformer output and other recent features to predict short-term BG (e.g. 5min later)
- This two-stage architecture allows better handling of both sequential context and individual patient variability.



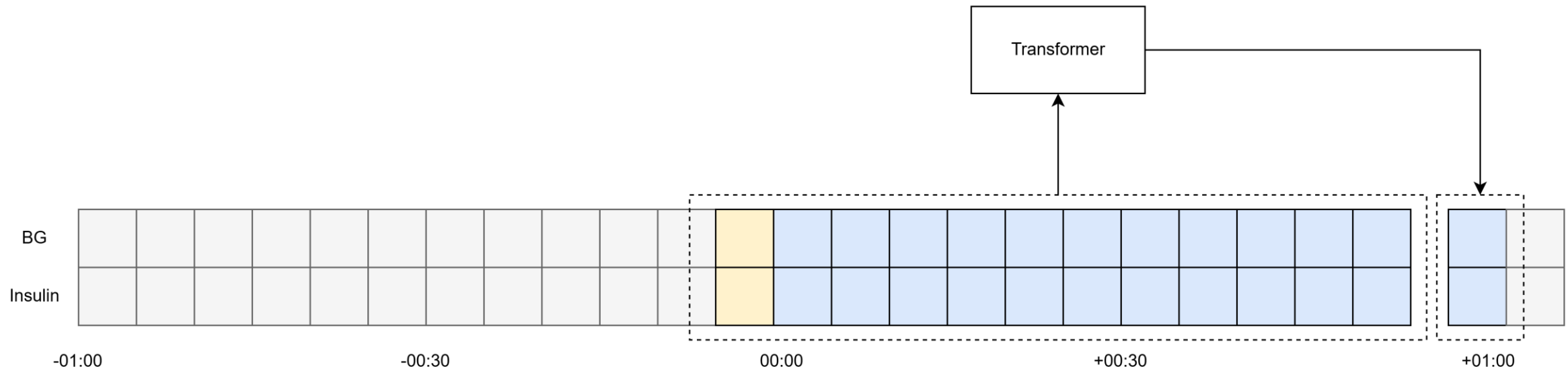
4. Future plan and the expected outcomes.



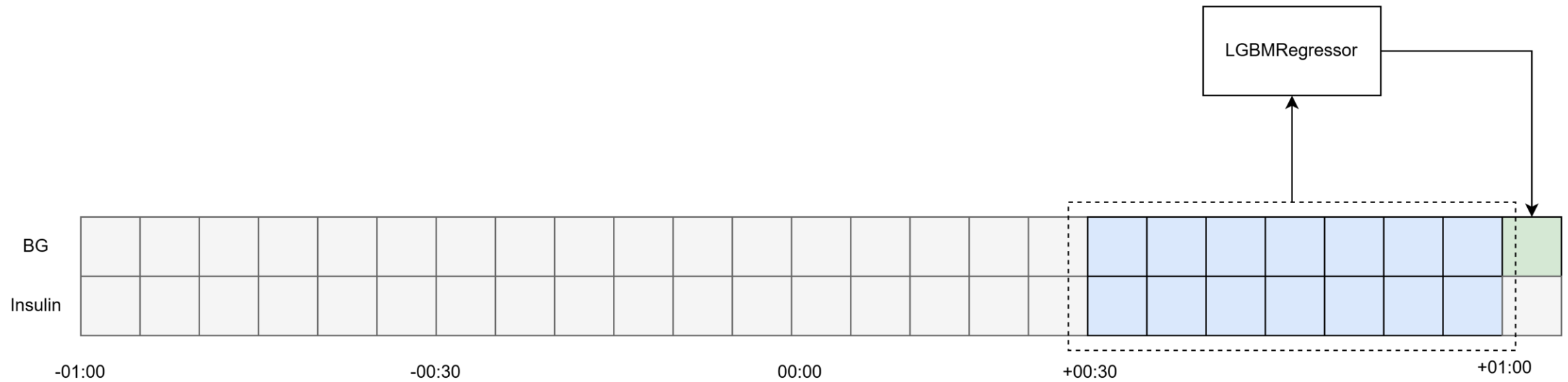
4. Future plan and the expected outcomes.



4. Future plan and the expected outcomes.

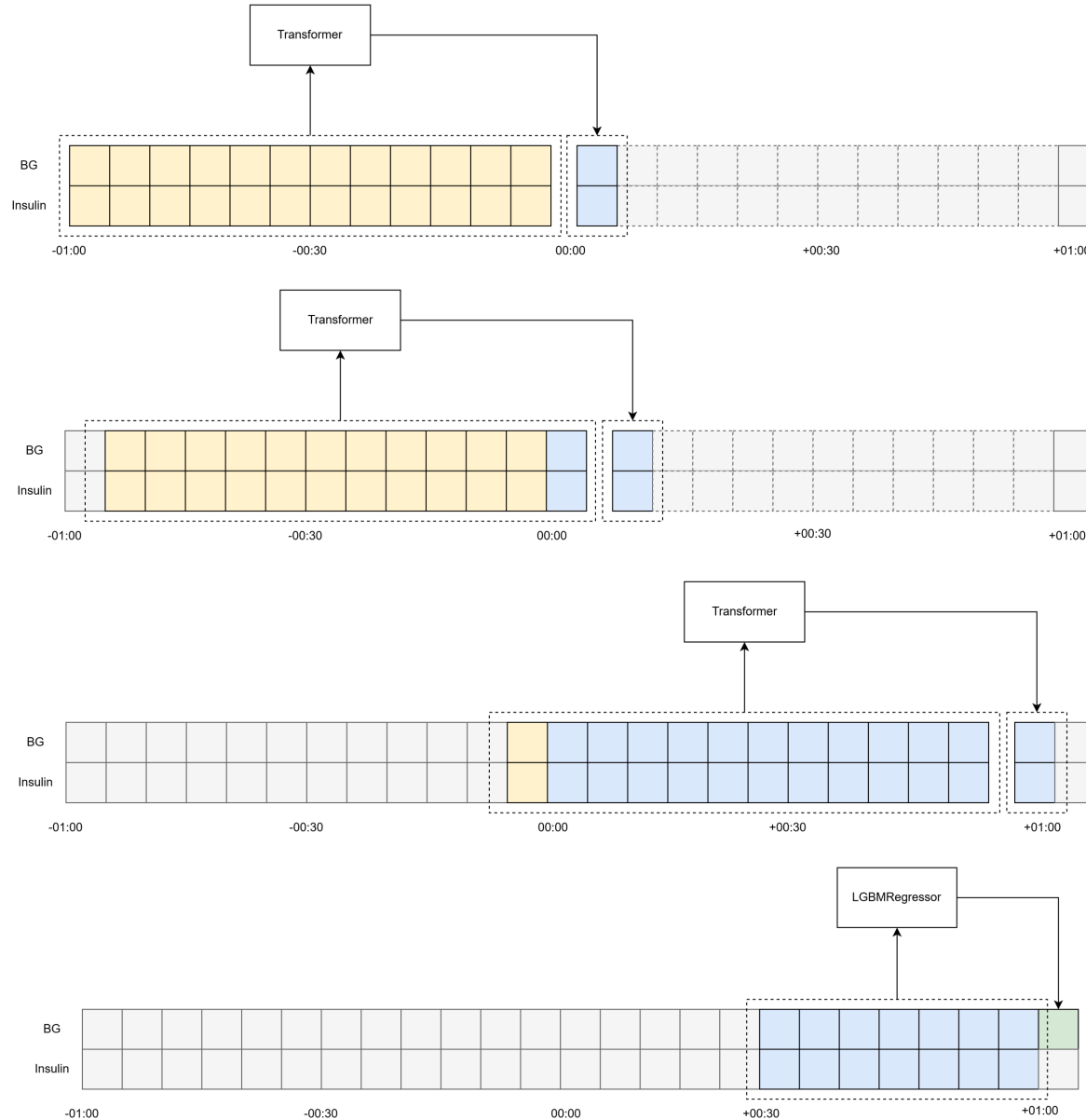


4. Future plan and the expected outcomes.



4. Future plan and the expected outcomes.

Transformer



Tree-based
Regressor