DataScientest Project - Brist1D

Contents

Overview

Predicting blood glucose fluctuations is crucial for managing type 1 diabetes. Developing effective algorithms for this can alleviate some of the challenges faced by individuals with the condition.

Team

- Svetlana Grinman (@svetigreen)
- Ralf Junghanns (@rabbl)
- Mathias Andres Saver Cortes (@masaver)
- Christian Weiland (@sander-NULL)

References

BrisT1D Blood Glucose Prediction Competition

Table of Contents

Context and Scope

In this section, we will define the context and scope of the project. We will cover the following topics:

- The kaggle challenge description
- What is type 1 diabetes and why is it important to predict blood glucose fluctuations?
- A general overview over the approach to blood glucose prediction with literature review

Kaggle Challenge

Predicting blood glucose fluctuations is a vital aspect of managing type 1 diabetes, as it helps improve treatment outcomes and daily quality of life. This project, inspired by a Kaggle Challenge, focuses on developing advanced algorithms to better anticipate glucose level changes, addressing the challenges faced by individuals living with this condition.

Goal

Forecast blood glucose levels one hour ahead using the previous six hours of participant data.

Description

Type 1 Diabetes

Type 1 diabetes is a chronic condition in which the body no longer produces the hormone insulin, making it unable to regulate blood glucose (sugar) levels. Without careful management, this can become life-threatening, therefore, the patients with this condition must inject insulin to manage their blood glucose levels. Many different factors, including eating, physical activity, stress, illness, sleep, alcohol, and many more, impact blood glucose levels, making insulin dosage calculations complex. The constant need to consider how an action may impact blood glucose levels and how to counteract them places a significant burden for those with type 1 diabetes.

An essential part of managing type 1 diabetes is predicting how blood glucose levels will change over time. While various algorithms have been developed for this purpose, the untidy nature of health data measurements and numerous unmeasured factors limit their effectiveness and accuracy. This competition aims to advance this work by challenging participants to predict future blood glucose using a newly collected dataset.

The Dataset

The dataset used in this competition is part of a newly collected, real-world data from young adults in the UK whith type 1 diabetes. All participants used continuous glucose monitors and insulin pumps, and a smartwatch was provided to collect activity data during the study. The complete dataset will be published after the competition for research purposes. Additional details about the study can be found in this blog post.

Evaluation

Submissions are evaluated on Kaggle based on Root Mean Square Error (RMSE) between the predicted blood glucose levels an hour into the future and the actual values collected at that time.

RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

where $\hat{y}\ i$ is the i-th predicted value, yi is the i-th true value and n is the number of samples.

The RMSE is computed from the `bg+1:00` (future blood glucose) predictions in the submission file, comparing them to the actual future blood glucose values. For both public and private

leaderboards, RMSE values are calculated using unknown, non-overlapping samples from the submission file across all participants.

Diabetes

Diabetes is a chronic condition that occurs when the pancreas no longer produces insulin, or the body cannot use insulin effectively.

Insulin is a hormone produced by the pancreas that acts like a key to allow glucose from the food we eat to pass from the bloodstream into the body's cells to generate energy. The body breaks down all carbohydrate foods into glucose in the blood, and insulin helps glucose move into the cells.

When the body is unable to produce or use insulin effectively, blood glucose levels rise, leading to hyperglycaemia. Prolonged high glucose levels can cause damage to the body and result in the failure of various organs and tissues.

There are several types of diabetes exist, the two most common are type 1 and type 2. This project specifically focuses on type 1 diabetes.

Diabetes Type 1

Type 1 diabetes is an autoimmune condition where the body attacks the insulin-producing cells in the pancreas. It is typically diagnosed in children and young adults and was previously known as juvenile diabetes. Individuals with type 1 diabetes must take insulin dayly to survive.

Blood Glucose

Blood glucose levels indicate the amount of glucose in the blood and are typically measured in millimoles per litre (mmol/L) or milligrams per decilitre (mg/dL). These levels are usually measured before meals, after meals, and at other various times throughout the day.

Managing blood glucose levels is a key part of managing diabetes. Individuals with diabetes need to maintain their blood glucose levels within a target range to minimize the risk of complications.

Known Complications of Unmanaged Diabetes

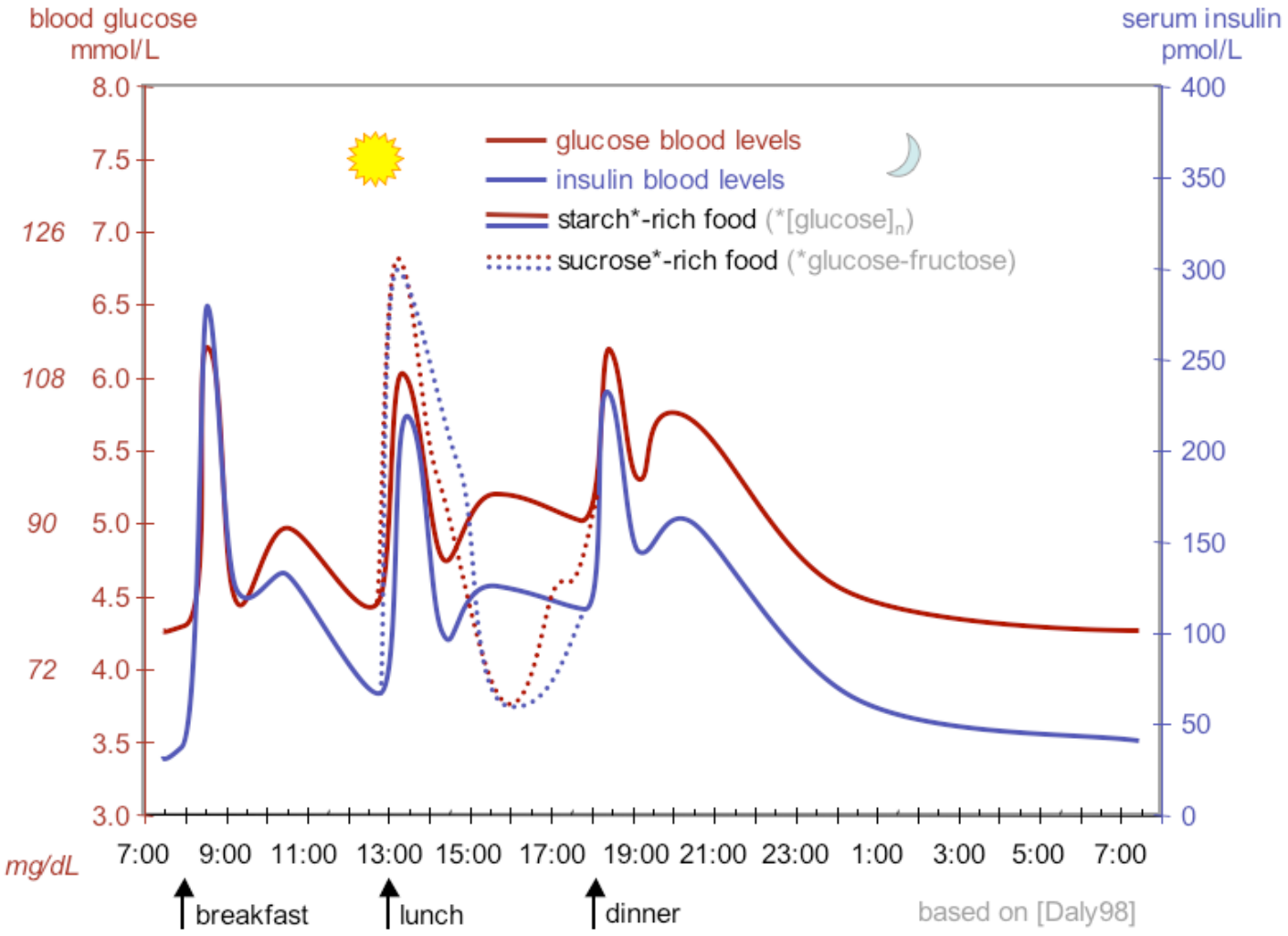
- Hyperglycemia (high blood sugar)
  - Fatigue, dehydration, increased thirst, frequent urination, blurred vision, and headaches.
- Hypoglycemia (low blood sugar)
  - Symptoms include shakiness, dizziness, sweating, hunger, irritability, headache, and blurred vision.

Long-term complications of unmanaged diabetes include:

- Heart disease
- Kidney disease
- Nerve damage
- Eye damage
- Foot damage

Blood Glucose Fluctuation over a Day [1]

The following figure illustrates typical blood glucose level fluctuations throughout the day, influenced by meals, physical activity, and other factors. The accompanying chart highlights the patterns and variability that can occur in glucose levels, emphasizing the importance of monitoring and maintaining a healthy range.



Target Blood Glucose Levels and Ranges in Type 1 Diabetes [2]

The following figure shows the recommended target blood glucose levels for people with type 1 diabetes. For this type, the target range is usually between 4.0 and 7.0 mmol/L before meals and less than 9 mmol/L 90 minutes after meals.

NICE recommended target blood glucose level ranges

Target Levels by Type	Upon waking	Before meals (pre prandial)	At least 90 minutes after meals (post prandial)
Non-diabetic*		4.0 to 5.9 mmol/L	under 7.8 mmol/L
Type 2 diabetes		4 to 7 mmol/L	under 8.5 mmol/L
Type 1 diabetes	5 to 7 mmol/L	4 to 7 mmol/L	5 to 9 mmol/L
Children w/ type 1 diabetes	4 to 7 mmol/L	4 to 7 mmol/L	5 to 9 mmol/L

*The non-diabetic figures are provided for information but are not part of NICE guidelines.

Factors Influencing Blood Glucose Levels [3]

The following figure lists various factors that can impact blood glucose levels. Upward arrows indicate factors that generally increase blood glucose, while sideways arrows represent a neutral effect.

42 Factors *that affect* Blood Glucose

FOOD

- ↑↑ 1 Carbohydrate quantity
- →↑ 2 Carbohydrate type
- →↑ 3 Fat
- →↑ 4 Protein
- →↑ 5 Caffeine
- ↓↑ 6 Alcohol
- ↓↑ 7 Meal timing
- ↑ 8 Dehydration
- ? 9 Personal microbiome

MEDICATION

- →↓ 10 Medication dose
- ↓↑ 11 Medication timing
- ↓↑ 12 Medication interactions
- ↑↑ 13 Steroid administration
- ↑ 14 Niacin (Vitamin B3)

ACTIVITY

- →↓ 15 Light exercise
- ↓↑ 16 High-intensity & moderate exercise
- →↓ 17 Level of fitness/training
- ↓↑ 18 Time of day
- ↓↑ 19 Food and insulin timing

The arrows show the general effect these 42 factors seem to have on blood glucose based on scientific research and/or our experiences at diaTribe. However, not every individual will respond in the same way, so the best way to see how a factor affects you is through your own data: check your blood glucose

BIOLOGICAL

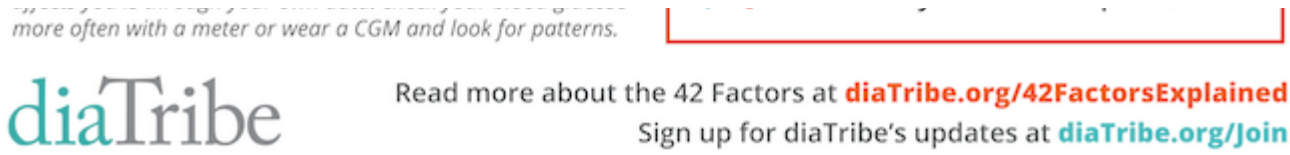
- ↑ 20 Too little sleep
- ↑ 21 Stress and illness
- ↓ 22 Recent hypoglycemia
- →↑ 23 During-sleep blood sugars
- ↑ 24 Dawn phenomenon
- ↑ 25 Infusion set issues
- ↑ 26 Scar tissue / lipodystrophy
- ↓↓ 27 Intramuscular insulin delivery
- ↑ 28 Allergies
- ↑ 29 A higher BG level (glucotoxicity)
- ↓↑ 30 Periods (menstruation)
- ↑↑ 31 Puberty
- ↓↑ 32 Celiac disease
- ↑ 33 Smoking

ENVIRONMENTAL

- ↑ 34 Expired insulin
- ↓↑ 35 Inaccurate BG reading
- ↓↑ 36 Outside temperature
- ↑ 37 Sunburn
- ? 38 Altitude

BEHAVIOR & DECISIONS

- ↓ 39 More frequent BG checks
- ↓↑ 40 Default options and choices
- ↓↑ 41 Decision-making biases
- ↓↑ 42 Family and social pressures



Responses to factors influencing blood glucose levels can vary significantly between individuals and even within the same person over time. Certain factors may have different impacts depending on whether a person has type 1 or type 2 diabetes. Factors marked with upward and downward arrows are particularly challenging to manage. The most reliable way to understand how a factor affects your blood glucose is through personal experience and observations.

The importance of Blood Glucose Prediction

Predicting blood glucose levels is essential for effective management of type 1 diabetes.

Anticipating changes in blood glucose over the next hour allows individuals to make informed decisions about insulin dosages, dietary choices, and physical activity. Accurate predictions enable proactive adjustments, helping to prevent dangerous highs or lows.

Maintaining blood glucose levels within the target range can help reduce the risk of complications associated with diabetes.

Literature Review

Introduction

This literature review provides an overview of the existing research on blood glucose prediction. One of the most import source for this review is a meta study [4] that provides an in-depth examination of various data-driven models for predicting blood glucose levels (BG) in people with type 1 diabetes. Accurate BG prediction is critical for improving diabetes management, helping patients avoid dangerous fluctuations in blood glucose levels, and reducing long-term health complications.

Other studies have also explored different approaches to BG prediction, including machine learning models, statistical methods, and physiological models. These studies have highlighted the challenges of BG prediction, such as the high variability of blood glucose levels, the complexity of the underlying physiological processes, and the need for personalized models that account for individual differences in patients.

Blood Glucose Prediction Model Approaches

Traditionally, prediction models for BGL have been based on physiological models that rely on extensive knowledge of the body's processes. However, data-driven approaches, which do not require detailed physiological understanding, have gained prominence.

These model approaches can be classified into three categories:

- time series forecasting (TSF),
- machine learning (ML),
- deep neural networks (DNN)

Despite their promise, comparing the performance of these models has been difficult due to differences in datasets, input variables, and model configurations.

In this meta study released in 2024 in Nature, the authors are comparing the effectiveness of these three approaches using:

- univariate (BG only)
- multivariate inputs (BG, carbohydrate intake, insulin dosages and physical activity)

Data Sources

The data used in these studies typically come from continuous glucose monitoring (CGM) devices, which provide real-time blood glucose measurements.

A very commonly used data set to compare models and inputs is the OhioT1DM dataset [5], which is publicly available and contains data from 30 patients with type 1 diabetes.

To outline the comparison of the three approaches, the authors of the meta study used the OhioT1DM dataset [5] to evaluate the performance of different models in predicting blood glucose levels.

Results

Model comparison

The performance of TSF, ML, and DNN models was compared based on their ability to predict blood glucose levels using univariate and multivariate inputs.

- TSF Model (ARIMA): This model showed stable but relatively low performance compared to the ML and DNN models. The ARIMA model struggled with multivariate input and often performed better when using only BGL data. This suggests that classical time series models may not be well-suited for handling complex, multivariate data, especially when incorporating exogenous variables like insulin and carbohydrate intake.
- ML Model (SVR): The SVR model consistently outperformed the other models, particularly when using multivariate inputs. It was also the fastest to train, making it a practical choice for real-time BGL prediction. The TML model's ability to integrate additional data like insulin dosage and physical activity proved beneficial in improving prediction accuracy, particularly for longer prediction horizons (60 minutes).
- DNN Model (LSTM): While the DNN model showed promising results, its performance was not significantly better than the TML model. It was also the slowest to train, which may limit its practicality for real-time applications. Interestingly, the DNN model performed similarly whether using univariate or multivariate inputs, suggesting that it might not fully utilize the additional data as effectively as the TML model.

Input Comparison

The comparison of univariate and multivariate inputs yielded mixed results.

- Univariate Input: Using only BG data for prediction performed well for all models, especially for short-term predictions (30 minutes). This is consistent with previous research suggesting that continuous glucose monitoring (CGM) data alone is sufficient for practical BGL prediction in real-world settings.
- Multivariate Input: While the ML model benefited from the inclusion of additional variables (carbohydrates, insulin, physical activity), the other models did not show significant improvements. In fact, adding multivariate data sometimes degraded the performance of the ARIMA model. The DNN model's performance remained largely unchanged, regardless of input type, implying that more advanced techniques for integrating multivariate data might be necessary to fully exploit its potential.

Discussion

The authors of the study highlight several important findings. First, ML models (particularly SVR) are highly effective for BG prediction, especially when additional data is available. Second, while multivariate input can improve prediction performance, especially for ML models, simply adding more variables does not guarantee better results. Advanced data fusion methods may be required to fully utilize multivariate inputs.

The study also emphasizes the practical implications of model selection. TML models, which are faster to train and perform well with multivariate data, could be more suitable for real-time applications, such as automated insulin delivery systems or continuous glucose monitoring devices.

Conclusion

The comprehensive analysis provided by this study offers valuable insights into the performance of different data-driven models for blood glucose prediction in T1DM. The findings suggest that ML models, particularly those using multivariate inputs, may offer the best balance of accuracy and speed for real-time BGL prediction. However, further research is needed to explore advanced techniques for integrating multivariate data, as well as the potential of hybrid models that combine data-driven and physiological approaches.

Exploratory Data Analysis

In this section, we will examine the raw dataset to gain insights into its structure, consistency, and key characteristics. This analysis aims to identify patterns, detect anomalies, and evaluate data quality, which are critical steps in preparing the dataset for further modelling and analysis.

Dataset Description and Structure

- Dataset description and structure
- Dataset consistency checks
    - Compare training and test data
    - Count data points for each patient

    - Time resolution and consistency
    - Lag-features consistency
- Data validation and cleaning
    - Outliers and anomalies
    - Missing values and imputation strategies
    - Further dataset discrepancies
- Data visualization
    - Time series data visualization
    - Lag-features visualization
    - Target variable visualization
- Data correlation
    - Correlation between lag-features and target variable
- Data distribution
    - Distribution of the target variable
    - Distribution of the lag-features

Dataset Description and Structure

Dataset Description

Dataset Description

The dataset originates from a study conducted on young adults in the UK with type 1 diabetes. Participants used a continuous glucose monitor (CGM), an insulin pump, and a smartwatch. These devices collected data, including blood glucose readings, insulin dosage, carbohydrate intake, and activity metrics. The collected data was aggregated into five-minute intervals and formatted into samples, each representing a point in time with aggregated data from the previous six hours. The goal is to predict the blood glucose level an hour into the future for each sample.

Training Set:

- Comprises samples from the first three months of the study for nine participants.
- Includes future blood glucose values for model training.
- Samples are presented in chronological order and overlap.

Testing Set:

- Consists of samples from the remaining study period for fifteen participants (unseen in the training set).
- Samples do not overlap and are randomized to prevent data leakage.

Complexities to consider:

- **Missing values and noise:** As with most medical datasets, there are incomplete and noisy data points.
- **Device variations:** Participants used different CGM, insulin pump, and smartwatch models, potentially introducing variability in data collection methods.
- **Unseen participants:** Some participants in the test set do not appear in the training set, posing an additional challenge for prediction models.

Columns

#Column	Name	Description	Type
1	id	row id consisting of participant number and a count for that participant	string
2	p_num	participant number	string
3	time	time of day in the format HH:MM:SS	string
4-75	bg-X:XX	blood glucose reading in mmol/L, X:XX(H:MM) time in the past	float
76-147	insulin-X:XX	total insulin dose received in units in the last 5 minutes, X:XX(H:MM) time in the past	float
148-219	carbs-X:XX	total carbohydrate value consumed in grammes in the last 5 minutes, X:XX(H:MM) time in the past	float
220-291	hr-X:XX	mean heart rate in beats per minute in the last 5 minutes, X:XX(H:MM) time in the past	float
292-363	steps-X:XX	total steps walked in the last 5 minutes, X:XX(H:MM) time in the past	float
364-435	cals-X:XX	total calories burnt in the last 5 minutes, X:XX(H:MM) time in the past	string
436-507	activity-X:XX	self-declared activity performed in the last 5 minutes, X:XX(H:MM) time in the past	string
508	bg-X:XX+1	blood glucose reading in mmol/L, X:XX+1(H:MM) time in the future, not provided in test.csv	float

Context of the Problem

Before starting, it is essential to define the context in which the problem is being addressed. Our objective is straightforward: we aim to predict a patient's blood glucose level, which is a continuous quantitative variable, based on other mostly continuous quantitative variables, for a point one hour

into the future. Given the nature of the target variable and the features, it is evident that we are dealing with a **supervised regression problem**.

Dataset Structure

```
from datetime import datetime

import matplotlib.pyplot as plt
import os
import pandas as pd
import seaborn as sns
```

Compare the Training and Test Data

We want to check if there are any differences between the two datasets columns.

```
# read train and test data
df_train = pd.read_csv(os.path.join('..', '..', '..', 'data', 'raw', 'train.csv'),
df_test = pd.read_csv(os.path.join('..', '..', '..', 'data', 'raw', 'test.csv'), lo
```

```
set(df_train.columns) - set(df_test.columns)
print(f'Columns in train but not in test: {list(set(df_train.columns) - set(df_test
```

```
Columns in train but not in test: ['bg+1:00']
```

```
df_train.head()
```

	id	p_num	time	bg-5:55	bg-5:50	bg-5:45	bg-5:40	bg-5:35	bg-5:30	bg-5:25	...	activity-0:4(
0	p01_0	p01	06:10:00	NaN	NaN	9.6	NaN	NaN	9.7	NaN	...	NaN
1	p01_1	p01	06:25:00	NaN	NaN	9.7	NaN	NaN	9.2	NaN	...	NaN
2	p01_2	p01	06:40:00	NaN	NaN	9.2	NaN	NaN	8.7	NaN	...	NaN
3	p01_3	p01	06:55:00	NaN	NaN	8.7	NaN	NaN	8.4	NaN	...	NaN
4	p01_4	p01	07:10:00	NaN	NaN	8.4	NaN	NaN	8.1	NaN	...	NaN

5 rows × 508 columns

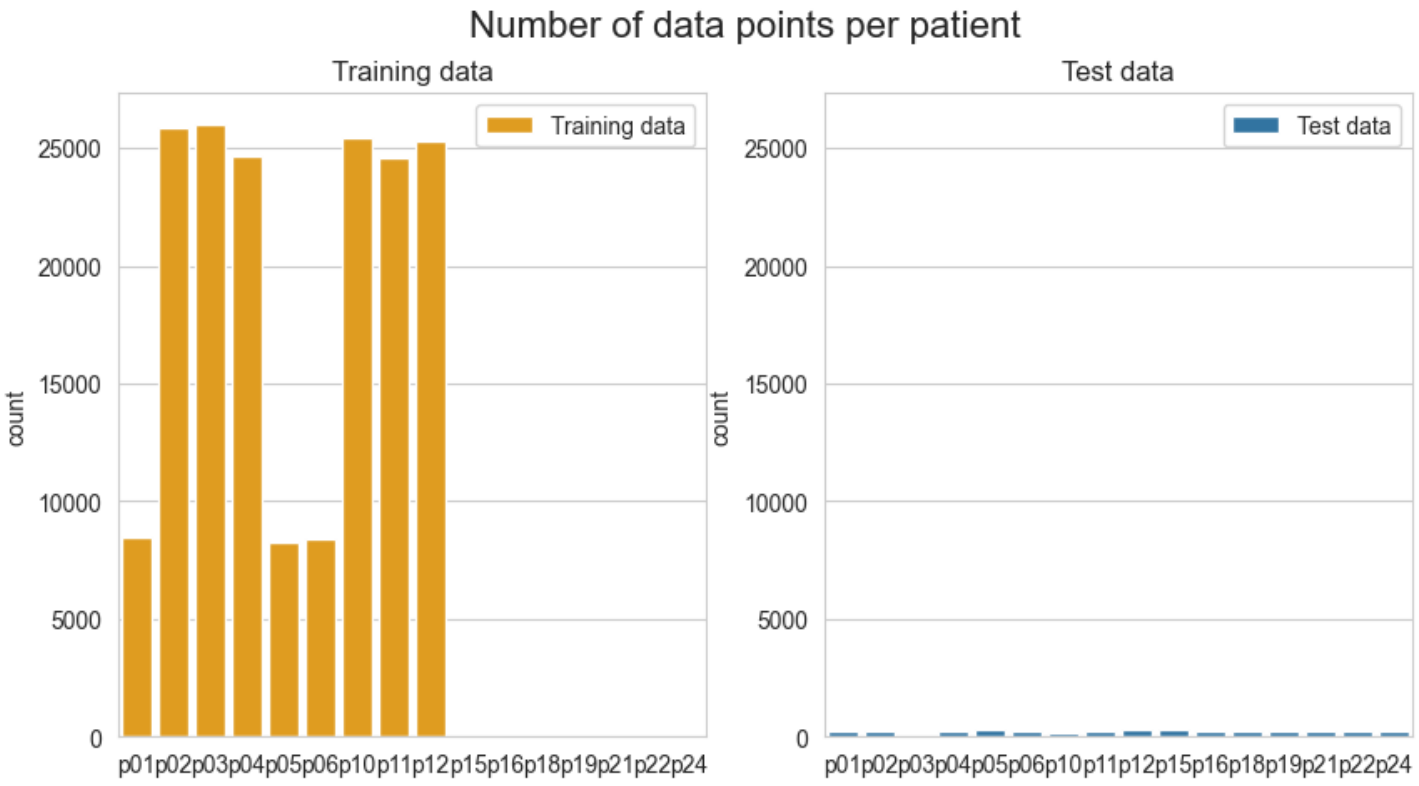
Number of Patients

```
print(f'Number of patients in training data: {len(df_train["p_num"].unique())}')
print(f'Number of patients in test data: {len(df_test["p_num"].unique())}')
```

```
Number of patients in training data: 9
Number of patients in test data: 15
```

Number of Datapoints per Patient

```
patient_list = sorted(list(set(df_train["p_num"].unique()) | set(df_test["p_num"].u
train_counts = df_train["p_num"].value_counts().reindex(patient_list, fill_value=0)
test_counts = df_test["p_num"].value_counts().reindex(patient_list, fill_value=0)

fig = plt.figure(figsize=(10, 5))
fig.suptitle("Number of data points per patient", fontsize=16)
ax1 = fig.add_subplot(1, 2, 1)
sns.barplot(x=patient_list, y=train_counts, label="Training data", color="orange")
ax1.set_title("Training data")

ax2 = fig.add_subplot(1, 2, 2)
sns.barplot(x=patient_list, y=test_counts, label="Test data")
ax2.set_title("Test data")

# Find the maximum y-axis limit between both plots
y_max = max(ax1.get_ylim()[1], ax2.get_ylim()[1])

# Set the same y-axis limit for both plots
ax1.set_ylim(0, y_max)
ax2.set_ylim(0, y_max)

plt.show();
```



Time Resolution per Patient in Training Data

```
patients_train = df_train["p_num"].unique()
patients_train
```

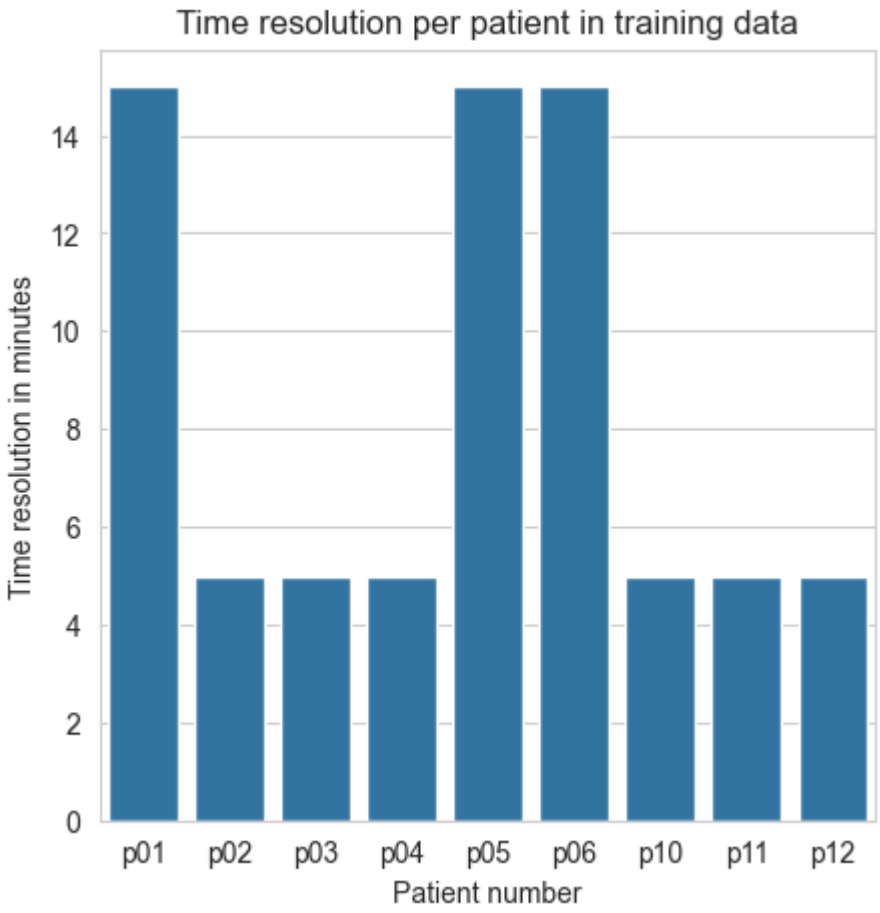
```
array(['p01', 'p02', 'p03', 'p04', 'p05', 'p06', 'p10', 'p11', 'p12'],
      dtype=object)
```

```
resolutions = pd.DataFrame(columns=["p_num", "time_resolution_in_minutes"])
for patient in patients_train:
    df_patient = df_train[df_train["p_num"] == patient]
    df_patient.loc[:, "time"] = pd.to_datetime(df_patient["time"], format="%H:%M:%S

    # Convert time objects to datetime objects (you can choose any date)
    datetime1 = datetime.combine(datetime.today(), df_patient["time"].iloc[0])
    datetime2 = datetime.combine(datetime.today(), df_patient["time"].iloc[1])

    print(datetime1, datetime2)

    # Subtract the datetime objects to get a timedelta
    time_difference = datetime2 - datetime1

    resolutions = pd.concat([
        resolutions,
        pd.DataFrame({"p_num": [patient], "time_resolution_in_minutes": [int(time_d
    ], ignore_index=True)

resolutions.head()
```

```
2024-10-22 06:10:00 2024-10-22 06:25:00
2024-10-22 06:05:00 2024-10-22 06:10:00
2024-10-22 06:05:00 2024-10-22 06:10:00
2024-10-22 06:05:00 2024-10-22 06:10:00
2024-10-22 06:05:00 2024-10-22 06:20:00
2024-10-22 10:25:00 2024-10-22 10:40:00
2024-10-22 06:05:00 2024-10-22 06:10:00
2024-10-22 06:05:00 2024-10-22 06:10:00
2024-10-22 10:25:00 2024-10-22 10:30:00
```

	p_num	time_resolution_in_minutes
0	p01	15
1	p02	5
2	p03	5
3	p04	5
4	p05	15

```
plt.figure(figsize=(5, 5))
sns.barplot(x="p_num", y="time_resolution_in_minutes", data=resolutions)
plt.title("Time resolution per patient in training data")
plt.xlabel("Patient number")
plt.ylabel("Time resolution in minutes")
plt.show()
```



Dataset Consistency and Validation

This section will cover the following topics:

- Quality Control and Assurance
- Outliers and Anomalies
- Missing Values
- Further Dataset Difficulties and Biases

Quality Control and Assurance

At this point, we have a dataset that is ready for analysis. However, before we start the analysis, we need to ensure that the dataset is consistent and valid. This is a crucial step in the data analysis process, as it ensures that the results of the analysis are accurate and reliable.

The training dataset consists of daily time series for each patient. The rows are supposed to be ordered consecutively by time. However, since no explicit “day” column is provided, we must verify

that the rows are indeed sequential. To do this, the lag features columns are utilized.

Validation Approach:

- Compare the lag features in each row to the corresponding data in preceding rows.
- Identify and flag any gaps or inconsistencies in the time series data.

Expected Outcome:
If the lag features align correctly with the data in prior rows, we can assume that the dataset is sequential and free from significant gaps, validating its integrity for further analysis.

Implementation

To ensure the correctness and reliability of the validation algorithm, we implemented it in two independent ways, developed by different team members.

1. The first implementation:
    - performs a cell-by-cell comparison of the lag feature columns,
    - identifies any changes in data types or values within the lag features.
2. The second implementation:
    - shifts and reindexes the lag feature columns based on the time differences.
    - detects non-unique values in the parameter columns for each row, effectively spotting discrepancies in sequential data.

Both implementations together enhance the robustness of the validation process and reduce the risk of undetected errors in the dataset.

Load the Dataset

```
import pandas as pd
import numpy as np
import os

# Load the dataset
df = pd.read_csv(os.path.join('..', '..', '..', 'data', 'raw', 'train.csv'), na_val
```

Define a Date Time Index

```
from helpers import set_datetime_index

df = set_datetime_index(df)
display(df.iloc[:, :4].head())
df = df.drop(columns=['time'])
```

	id	p_num	time	bg-5:55
datetime				
2000-01-01 06:10:00	p01_0	p01	06:10:00	NaN
2000-01-01 06:25:00	p01_1	p01	06:25:00	NaN
2000-01-01 06:40:00	p01_2	p01	06:40:00	NaN
2000-01-01 06:55:00	p01_3	p01	06:55:00	NaN
2000-01-01 07:10:00	p01_4	p01	07:10:00	NaN

Consistency Check for Lag Features of Parameters and the Target Variable

```
from helpers import consistency_check

result_dict = consistency_check(df)
```

Display the Results

The following table summarizes the results of the validation process. It shows the number of rows where the lag features did not correspond to the data in the preceding rows, alongside the total number of rows for each patient and parameter. Additionally, the `target` column compares the `bg` parameter (blood glucose) with a time difference of +1:00.

```
from helpers import get_parameters

result = pd.DataFrame.from_dict(result_dict, columns=get_parameters() + ['total'],
result = result.loc[:, (result != 0).any(axis=0)]
result
```

	hr	steps	cals	total
p01	0	0	0	16865
p02	0	0	0	26335
p03	0	0	0	26427
p04	0	0	0	25047
p05	0	0	0	16248
p06	0	0	0	16674
p10	0	0	0	25874
p11	68	26	45	25205
p12	0	0	0	26048

Conclusion

The analysis reveals that the lag features are largely consistent with the data in the preceding rows, indicating a high level of dataset reliability. However, there are minor inconsistencies observed in Patient `p11`, particularly in the parameters `heartrate`, `steps`, `cals` due to overlapping values in time. These values can be fixed by shifting the datetime index `+1day` at index `p11_4307`.

No more data inconsistencies were found. The `target` column also shows no discrepancies.

Outliers and Anomalies

In this notebook, we will analyze the dataset for outliers and anomalies, as addressing them is a crucial step in **standardizing** numerical values during the implementation of a machine learning project.

Given that the explanatory variables represent time series, we will treat the group of column series (e.g., `col-*`) as a single variable type and inspect them in detail.

```
import pandas as pd
import os

base_dir = os.path.abspath(os.path.join('..', '..', '..', 'data', 'raw'))
file_path = os.path.join(base_dir, 'train.csv')

patients = pd.read_csv(file_path, low_memory=False)
```

```
bg_cols = [col for col in patients.columns if col.startswith('bg-')]
insulin_cols = [col for col in patients.columns if col.startswith('insulin-')]
carbs_cols = [col for col in patients.columns if col.startswith('carbs-')]
hr_cols = [col for col in patients.columns if col.startswith('hr-')]
steps_cols = [col for col in patients.columns if col.startswith('steps-')]
cals_cols = [col for col in patients.columns if col.startswith('cals-')]
activity_cols = [col for col in patients.columns if col.startswith('activity-')]
```

> *Note:* The commonly used IQR method for outlier detection is not applicable to our dataset due to its skewed distribution. This issue will be addressed in the next phase of the project - “Data Preprocessing”.

Outliers for bg-* columns

```
patients[bg_cols].describe()
```

	bg-5:55	bg-5:50	bg-5:45	bg-5:40	bg-5:35
count	149770.000000	158533.000000	163364.000000	149766.000000	158254.000000
mean	8.211018	8.230449	8.253291	8.210988	8.229649
std	2.852188	2.913438	2.945594	2.852090	2.911313
min	2.200000	2.200000	2.200000	2.200000	2.200000
25%	6.100000	6.100000	6.100000	6.100000	6.100000
50%	7.600000	7.600000	7.700000	7.600000	7.600000
75%	9.800000	9.800000	9.800000	9.800000	9.800000
max	22.200000	25.100000	27.800000	22.200000	25.100000

8 rows × 72 columns

```
# Get the describe() result for all 'bg-*' columns
df_bg = patients[bg_cols].describe()

# Extract the minimum and maximum values across all bg-* columns
overall_min = df_bg.loc['min'].min()  # Get the smallest 'min' value across all bg-
overall_max = df_bg.loc['max'].max()  # Get the largest 'max' value across all bg-*

# Display the results
print(f"Overall minimum value across all bg-* columns: {overall_min}")
print(f"Overall maximum value across all bg-* columns: {overall_max}")
```

```
Overall minimum value across all bg-* columns: 2.2
Overall maximum value across all bg-* columns: 27.8
```

```
patients.groupby('p_num')['bg-5:45'].agg(['min', 'max'])
```

	min	max
p_num		
p01	2.3	27.8
p02	2.2	22.2
p03	2.2	22.2
p04	2.2	18.4
p05	2.9	20.6
p06	2.9	27.8
p10	2.2	15.9
p11	2.2	20.8
p12	2.8	22.2

> **Summary**: There are some extreme values for blood glucose present but still realistic for some patients.

Outliers for insulin-* columns

```
# Get the describe() result for all 'insulin-*' columns
df_insulin = patients[insulin_cols].describe()

# Extract the minimum and maximum values across all insulin-* columns
overall_min = df_insulin.loc['min'].min()
overall_max = df_insulin.loc['max'].max()

# Display the results
print(f"Overall minimum value across all insulin-* columns: {overall_min}")
print(f"Overall maximum value across all insulin-* columns: {overall_max}")
```

```
Overall minimum value across all insulin-* columns: -0.3078
Overall maximum value across all insulin-* columns: 46.311
```

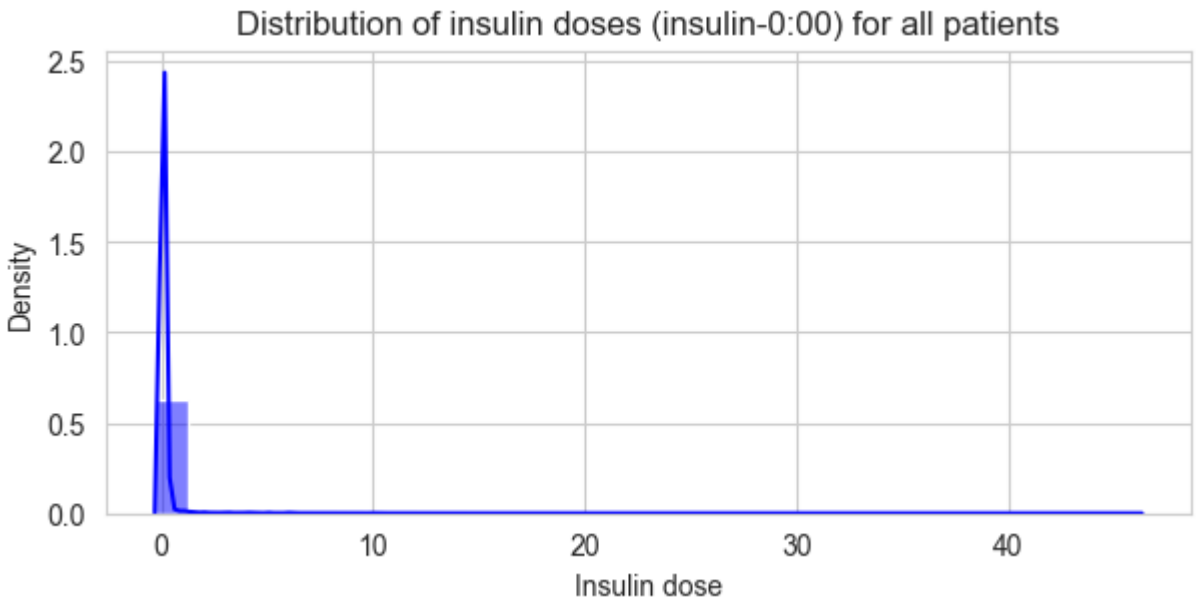
a) Investigating negative insulin

```
# (patients[insulin_cols] < 0).groupby(patients['p_num']).sum()  # result: only p12

# Create an empty list to store unique negative values
unique_negative_values = set()

# Iterate through each insulin column and patient 'p12'
for col in insulin_cols:
    negative_values = patients[(patients['p_num'] == 'p12') & (patients[col] < 0)][

    # Add the negative values to the set (automatically handles uniqueness)
    unique_negative_values.update(negative_values.dropna())

# Convert the set to a sorted list and display
unique_negative_values = sorted(unique_negative_values)
print(unique_negative_values)
```

```
[-0.3078, -0.059]
```

a) Investigating positiv extremes

```
import matplotlib.pyplot as plt
import seaborn as sns

insulin_values = patients["insulin-0:00"].dropna()

# Create a figure for the histogram and KDE plot
plt.figure(figsize=(7, 3))

sns.histplot(insulin_values, bins=30, kde=True, color='blue', stat='density')

# Add labels and title
plt.xlabel('Insulin dose')
plt.ylabel('Density')
plt.title('Distribution of insulin doses (insulin-0:00) for all patients')

plt.show()
```




```
patients.groupby('p_num')['insulin-0:00'].agg(['min', 'max'])
```

	min	max
p_num		
p01	0.0000	11.7417
p02	0.0000	18.0833
p03	0.0000	46.3110
p04	0.0000	42.7800
p05	0.0000	8.1542
p06	0.0000	14.1833
p10	0.0000	9.0833
p11	0.0000	12.4167
p12	-0.3078	25.3500

> **Summary:** For patient `p12`, we detected some negative values. Negative insulin values are often the result of data entry errors, sensor malfunctions, or recording issues in the dataset. Since there are only two different negative values and they fall outside the usual value ranges, we will replace them with either their corresponding positive values or zeros. This step will be carried out during the data preprocessing phase.
>
> Additionally, for two other patients, we observe extremely high insulin doses. While these values do not appear to be errors, the data is still right-skewed and will need to be addressed.

Outliers for carbs-* columns

```
# Get the describe() result for all 'carbs-*' columns
df_carbs = patients[carbs_cols].describe()

# Extract the minimum and maximum values across all hr-* columns
overall_min = df_carbs.loc['min'].min()
overall_max = df_carbs.loc['max'].max()

# Display the results
print(f"Overall minimum value across all carbs-* columns: {overall_min}")
print(f"Overall maximum value across all carbs-* columns: {overall_max}")
```

```
Overall minimum value across all carbs-* columns: 1.0
Overall maximum value across all carbs-* columns: 852.0
```

> **Summary:** The values for carbohydrate consumption, ranging from 1.0 to 852.0 grams, raise concerns about their realism and appropriateness for our use case. Given that these data are self-reported by patients and 98% of the values are missing, we may consider excluding these columns from the model.

Outliers for hr-* columns

```
# Get the describe() result for all 'hr-*' columns
df_hr = patients[hr_cols].describe()

# Extract the minimum and maximum values across all hr-* columns
overall_min = df_hr.loc['min'].min()
overall_max = df_hr.loc['max'].max()  # Group by 'p_num' and calculate the min and
patients.groupby('p_num')['hr-5:55'].agg(['min', 'max'])

# Display the results
print(f"Overall minimum value across all hr-* columns: {overall_min}")
print(f"Overall maximum value across all hr-* columns: {overall_max}")
```

```
Overall minimum value across all hr-* columns: 37.6
Overall maximum value across all hr-* columns: 185.3
```

```
# Group by 'p_num' and calculate the min and max of 'hr-5:55' for each patient
patients.groupby('p_num')['hr-5:55'].agg(['min', 'max'])
```

	min	max
p_num		
p01	47.3	163.8
p02	40.0	185.3
p03	47.2	156.8
p04	50.7	164.4
p05	43.9	158.2
p06	37.6	155.1
p10	47.4	184.2
p11	49.7	164.8
p12	49.7	136.6

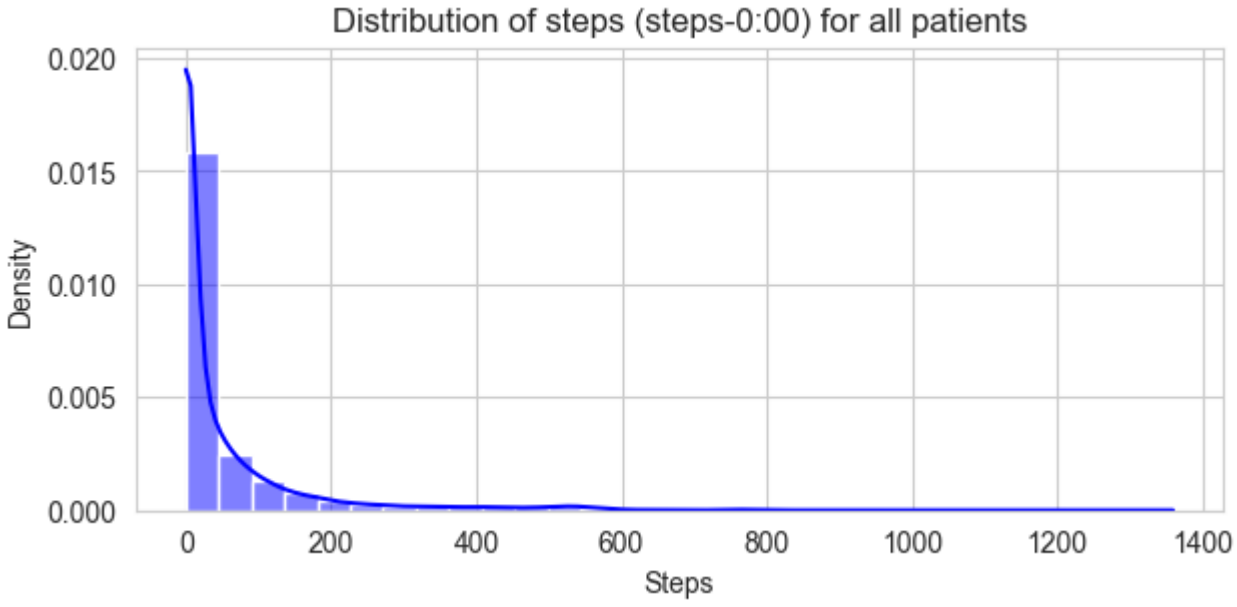
> **Summary:** For one patient (`p06`), we observed an unusually low heart rate of 37.6 bpm. While this value could be realistic for elite athletes or during deep sleep, it is generally considered abnormally low for most people. For now, we will leave this value as is, as it might indicate an important clinical signal (e.g., severe bradycardia). However, we should keep in mind that in a small dataset of only nine patients, even a single outlier can have a larger impact compared to larger datasets. If we apply models like random forests, gradient boosting or SVM - which are more robust to outliers - this outlier might have minimal impact.
>
> A similar consideration applies to the extreme value of 185.3 bpm for patient `p02`.

Outliers for steps-* columns

```
# Get the describe() result for all 'steps-*' columns
df_steps = patients[steps_cols].describe()

# Extract the minimum and maximum values across all steps-* columns
overall_min = df_steps.loc['min'].min()
overall_max = df_steps.loc['max'].max()

# Display the results
print(f"Overall minimum value across all steps-* columns: {overall_min}")
print(f"Overall maximum value across all steps-* columns: {overall_max}")
```

```
Overall minimum value across all steps-* columns: 0.0
Overall maximum value across all steps-* columns: 1359.0
```

```python
import matplotlib.pyplot as plt
import seaborn as sns

steps_values = patients["steps-0:00"].dropna()

# Create a figure for the histogram and KDE plot
plt.figure(figsize=(7, 3))

sns.histplot(steps_values, bins=30, kde=True, color='blue', stat='density')

# Add labels and title
plt.xlabel('Steps')
plt.ylabel('Density')
plt.title('Distribution of steps (steps-0:00) for all patients')

plt.show()
```




```python
# Group by 'p_num' and calculate the min and max of 'steps-0:00' for each patient
patients.groupby('p_num')['steps-0:00'].agg(['min', 'max'])
```

	min	max
p_num		
p01	0.0	741.0
p02	1.0	1359.0
p03	0.0	582.0
p04	0.0	673.0
p05	0.0	660.0
p06	0.0	627.0
p10	0.0	783.0
p11	0.0	660.0
p12	0.0	567.0

```
# Filter the dataset for patient 'p02' where 'steps-0:00' equals 1359
activity_steps_1359_p02 = patients[(patients['p_num'] == 'p02') & (patients['steps-

# Display the result
print(activity_steps_1359_p02)
```

```
      activity-0:00  steps-0:00
19127       Walking      1359.0
```

```
patients[(patients['p_num'] == 'p02') & (patients['steps-0:00'] == 1359)][steps_col
```

	steps-5:55	steps-5:50	steps-5:45	steps-5:40	steps-5:35	steps-5:30	steps-5:25	steps-5:20	steps-5:15	steps-5:1
19127	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na

1 rows × 144 columns

> **Summary:** The average walking speed for a healthy adult is around 100 to 120 steps per minute. Over a 5-minute period, a typical person would walk approximately 500 to 600 steps at a moderate pace. Therefore, a value of 1359 steps appears to be an outlier or extreme value. However, after considering the frequency and trends in neighboring records, along with

> the fact that the activity was reported as “Walking,” we can conclude that this value is likely valid in the context of possible intense physical activity.

Outliers for cals-* columns

```
# Get the describe() result for all 'cals-*' columns
df_cals = patients[cals_cols].describe()

# Extract the minimum and maximum values across all steps-* columns
overall_min = df_cals.loc['min'].min()
overall_max = df_cals.loc['max'].max()

# Display the results
print(f"Overall minimum value across all cals-* columns: {overall_min}")
print(f"Overall maximum value across all cals-* columns: {overall_max}")
```

```
Overall minimum value across all cals-* columns: 0.03
Overall maximum value across all cals-* columns: 116.1
```

> **Summary:** The values observed here appear realistic, considering potential physical activities.

Missing Values

In this notebook, we will analyze the dataset for missing values, as handling missing data is a critical aspect of ensuring the **completeness**, one of the fundamentals of **Data Quality**, during the implementation of a machine learning project.

Given that the explanatory variables represent time series, we will treat the group of column series (e.g., col-*) as a single variable type and inspect them in detail.

```
import pandas as pd
import os

base_dir = os.path.abspath(os.path.join('..', '..', '..', 'data', 'raw'))
file_path = os.path.join(base_dir, 'train.csv')

patients = pd.read_csv(file_path, low_memory=False)
patients.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 177024 entries, 0 to 177023
Columns: 508 entries, id to bg+1:00
dtypes: float64(433), object(75)
memory usage: 686.1+ MB
```

Percentage of missing values for bg-* columns

```
# select all the 'bg-*' columns except the target one
bg_cols = [col for col in patients.columns if col.startswith('bg-')]

# calculates the percentage of missing values for each bg-* column individually and
missing_percentage = patients[bg_cols].isnull().mean() * 100
print(min(missing_percentage), max(missing_percentage))
```

```
1.5229573391178597 15.399606832971802
```

```
missing_percentage = patients[bg_cols].isna().sum().sum() / patients[bg_cols].size
missing_percentage
```

```
np.float64(10.761911755844782)
```

Conclusion & handling: There are between 1.5% and 15% missing values across the bg-* columns. Given that these data are recorded by a continuous glucose monitor (CGM), with some patients having data recorded at 15-minute intervals, we assume that after proper interpolation and imputation, the missing values will be resolved. This step will be carried out during the data preprocessing phase.

Percentage of missing values for insulin-* columns

```
insulin_cols = [col for col in patients.columns if col.startswith('insulin-')]

# Calculate the percentage of missing values for each 'insulin-*' column individual
missing_percentage = patients[insulin_cols].isnull().mean() * 100
print(min(missing_percentage), max(missing_percentage))
```

```
5.295892082429502 5.335999638467101
```

Conclusion & handling: There are approximately 5.3% missing values accross the insulin-* columns. Given that these data are recorded by the insulin pump, with some patients having data recorded at 15-minute intervals, we assume that after proper interpolation and imputation, the missing values will be resolved. This step will be carried out during the data preprocessing phase.

Percentage of missing values for carbs-* columns

```
carbs_cols = [col for col in patients.columns if col.startswith('carbs-')]

# Calculate the percentage of missing values for each 'carbs-* column individually
missing_percentage = patients[carbs_cols].isnull().mean() * 100
print(min(missing_percentage), max(missing_percentage))
```

```
98.53918112798264 98.57194504699928
```

Conclusion & handling: There are approximately 98.5% missing values accross the carbs-* columns. Since these data are self-reported by participants and may not be reliable, we assume this variable will not significantly impact future predictions and can be dropped from the model.

Percentage of missing values for hr-* columns

```
hr_cols = [col for col in patients.columns if col.startswith('hr-')]

# Calculate the percentage of missing values for each 'hr-*' column individually an
missing_percentage = patients[hr_cols].isnull().mean() * 100
print(min(missing_percentage), max(missing_percentage))
```

```
28.885348879248014 29.272302060737527
```

```
patients[hr_cols].isna().sum().sum() / patients[hr_cols].size * 100
```

```
np.float64(29.10068377420262)
```

Conclusion & handling: There are approximately 29.1% missing values accross the hr-* columns. Given that these data are recorded by the smartwatch, with some patients having data recorded at 15-minute intervals, we assume that after proper interpolation and imputation, the missing values will be partially resolved. Any remaining missing values will be imputed using an appropriate method (to be determined). This step will be carried out during the data preprocessing phase.

Percentage of missing values for steps-* columns

```
steps_cols = [col for col in patients.columns if col.startswith('steps-')]

# Calculate the percentage of missing values for each 'steps-*' column individually
missing_percentage = patients[steps_cols].isnull().mean() * 100
print(min(missing_percentage), max(missing_percentage))
```

```
53.66165039768619 54.055382321041215
```

```
patients[steps_cols].isna().sum().sum() / patients[steps_cols].size * 100
```

```
np.float64(53.87635763135695)
```

Conclusion & handling: There are approximately 54% missing values accross the steps-* columns. Given that these data are recorded by the smartwatch, with some patients having data recorded at 15-minute intervals, we assume that after proper interpolation and imputation, the missing values will be partially resolved. Any remaining missing values will be replaced with 0 (to be confirmed). This step will be carried out during the data preprocessing phase.

Percentage of missing values for cals-* columns

```
cals_cols = [col for col in patients.columns if col.startswith('cals-')]

# Calculate the percentage of missing values for each 'cals-*' column individually
missing_percentage = patients[cals_cols].isnull().mean() * 100
print(min(missing_percentage), max(missing_percentage))
```

```
19.916508496023138 20.231155097613883
```

```
patients[cals_cols].isna().sum().sum() / patients[cals_cols].size * 100
```

```
np.float64(20.07893939051579)
```

Conclusion & handling: There are approximately 20% missing values accross the cals-* columns. Given that these data are recorded by the smartwatch, with some patients having data recorded at 15-minute intervals, we assume that after proper interpolation and imputation, the missing values will be partially resolved. Any remaining missing values will be replaced with 0. This step will be carried out during the data preprocessing phase.

Percentage of missing values for activity-* columns

```
activity_cols = [col for col in patients.columns if col.startswith('activity-')]
# Calculate the percentage of missing values for each 'activity-*' column individua
missing_percentage = patients[activity_cols].isnull().mean() * 100
print(min(missing_percentage), max(missing_percentage))

# Initialize an empty set to store unique activities
unique_activities = set()
for col in activity_cols:
    unique_activities.update(patients[col].dropna().unique())
unique_activities_list = list(unique_activities)
print(unique_activities_list)
```

```
98.43411062906723 98.46800433839479
['Hike', 'Zumba', 'Swim', 'HIIT', 'Strength training', 'Indoor climbing', 'Bike', 'I
```

Percentage of reported activities for each patient

```
# find the "busiest" patient, i.e. the one who reported the most activities across
# Count non-missing values in the 'activity-*' columns for each patient
activity_counts = patients.groupby('p_num')[activity_cols].apply(lambda x: x.notnul

# Find the patient with the maximum activity reports
busiest_patient = activity_counts.idxmax()
max_activities_reported = activity_counts.max()

# Output the result
print(f"The busiest patient is {busiest_patient} with {max_activities_reported} rep

# Note: this is the highest absolute number of activity entries
```

```
The busiest patient is p10 with 65952 reported activities.
```

```
# Count non-missing values in the 'activity-*' columns for each patient
activity_counts = patients.groupby('p_num')[activity_cols].apply(lambda x: x.notnul

# Calculate the total possible activity entries for each patient
total_possible_activities = len(activity_cols) * patients.groupby('p_num').size()

# Calculate the percentage of filled activity data for each patient
activity_percentage_filled = (activity_counts / total_possible_activities) * 100

# Display the percentage for each patient
print(activity_percentage_filled)

# Note: this is the highest percentage of filled activity data, which suggests that
# for this patient is filled in, even if the absolute number of activity records is
```

```
p_num
p01    3.935650
p02    0.935857
p03    0.430306
p04    1.124625
p05    1.233879
p06    2.589401
p10    3.598649
p11    1.884432
p12    0.304525
dtype: float64
```

Conclusion & handling: There are approximately 98.4% missing values across the activity-* columns. Although these data are self-reported by participants and may not be entirely reliable, we assume this variable might still be useful for model predictions. For at least two participants who

actively reported their activities, it would be interesting to evaluate whether the model shows significant improvements. If no substantial benefit is observed, this feature can be dropped. Further analysis on this is required.

Percentage of missing values for the target variable bg+1:00

```
patients['bg+1:00'].isna().sum() / len(patients) * 100
```

```
np.float64(0.0)
```

Conclusion & handling: There are no missing values in the bg+1:00 column, so no further action is required.

Further Dataset Difficulties and Biases

The objective of the first step in this learning project was to explore the dataset and conduct a thorough analysis to uncover its structure, challenges, and potential biases. This notebook highlights the most significant observations and considerations. While many important decisions cannot be made at this stage, they will be addressed in the subsequent steps of the project.

Observations on Data

- **Inconsistency:** different time intervals for different patients in the dataset (some recorded every 5 minutes and others every 15 minutes)
- **Potential data loss:** if we attempt to standardize the time intervals, we risk losing data or oversimplifying trends, especially in patients with finer granularity (5-minute intervals).

Possible Approaches:

1. Resampling the data to a consistent time interval:

- Up-sampling (resample to 5 minutes for all patients).
- Down-sampling (resample to 15 minutes for all patients).

a) Up-sampling: involves resampling all patient data to the finer 5-minute intervals. For patients who have data recorded every 15 minutes, we could interpolate missing values for the in-between

time points.

Pros:

- Keeps the finer-granularity data intact for those patients who already have 5-minute intervals.
- Allows for more detailed time-series analysis.

Cons:

- Interpolating data for the patients who originally have 15-minute intervals might introduce artificial data points, which may not capture the true variability of the measurements.

b) Down-sampling: resampling everyone's data to 15-minute intervals by aggregating the 5-minute data for patients with finer granularity (e.g. taking averages or sums over each 15-minute period).

Pros:

- Simpler, and no interpolation is needed.
- Keeps the data more consistent with what was actually measured for those with 15-minute intervals.

Cons:

- We might lose detail from patients who had more frequent measurements, which could result in losing important patterns in their data.

2. Handling each interval separately: another option for building models on patients with 5-min interval separately from the patients with 15-min interval.

Pros:

- We retain the original data for each patient.
- We don't need to interpolate or downsample keeping the true variability.

Cons:

- This requires more work since we're essentially running two analyses or models, one for each group.
- It leads to smaller training sets for each group.

3. Feature Engineering: we can create additional feature to account for differences in time intervals:

- Time interval: indicating whether the data point comes from a 5-min or 15-min interval.

This will let the model account for different time resolutions without explicitly resampling the data.

Recommended approach: Based on the fact that we have only 3 patients with a 15-minute interval and 6 patient with a 5-minute interval, it is reasonable to resample the dataset to 5-minute intervals for all patients. Steps:

- Resample the data for all patients to 5-minute intervals.
- Interpolate the missing data points for patients who originally had 15-minute intervals.

Challenges and Potential Difficulties

- **Missing Data**
  - In the dataset, many columns (e.g. carbs-*, activity-*) had high percentages of missing values. This is a common issue in studies involving self-reported data or continuous monitoring (device resets etc.).
  - Handling strategy: impute missing values, use of statistical models to fill in gaps.
- **Outliers**
  - From the variable analysis, we observed a large number of outliers, which were primarily caused by the skewness present in most features. As a result, the commonly used IQR method for outlier detection is not applicable in this case.
  - Handling strategy: consider transformations to reduce skewness before detecting outliers.
- **Non-normal distributions**
  - Features like insulin and blood glucose are often highly skewed. This could impact the performance of models that assume normality or require scaling.
- **Feature engineering for Time-Series**
  - Extracting useful features such as:
    - Lagged features: past glucose, insulin, carb or other values from previous time steps.
    - Rolling averages/windows: smoothed glucose or insulin averages over time to capture trends.
- **Multicollinearity**

- Since the dataset contains many related measurements over time, there might be strong correlations between features. This could impact model performance, especially for regression models.
- Handling strategy: regularization techniques (Lasso, Ridge) or dimensionality reduction (PCA).

Possible Biases

- **Selection Bias:**
  - If the study only included certain types of participants (e.g. individuals with a specific health condition, age group, or gender), this could lead to selection bias, meaning the results may not generalize to other populations.
- **Measurement Bias:**
  - Self-reported data like carbohydrate intake and activity are prone to inaccuracies, which can lead to measurement bias. Wearable devices (for heart rate, steps, etc.) can also be less reliable depending on usage conditions.
- **Sampling Bias:**
  - If the self-reported measurements are not taken consistently (e.g., some participants report their data more frequently than others), the dataset might not represent all participants equally, leading to biased conclusions.

Patient Time Series Overview

In this notebook, we explore the time series data for individual patients to better understand the patterns and variations within the dataset.

Objectives:

- Examine specific patients' time series data over a **day** and a **week**.
- Identify trends and fluctuations in key parameters such as blood glucose levels, heartrate, and other recorded metrics.

This analysis provides a clearer picture of how the dataset captures the daily and weekly variations in patient metrics, offering insights into the complexity of the collected data.

```
import os
import pandas as pd
import matplotlib.dates as mdates
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

data_folder = os.path.join('../../../data/interim')
all_data = pd.read_csv(os.path.join(data_folder, 'all_train.csv'), index_col=0)
all_data.index = pd.to_datetime(all_data.index)
all_data['day_of_year'] = all_data.index.day_of_year
```

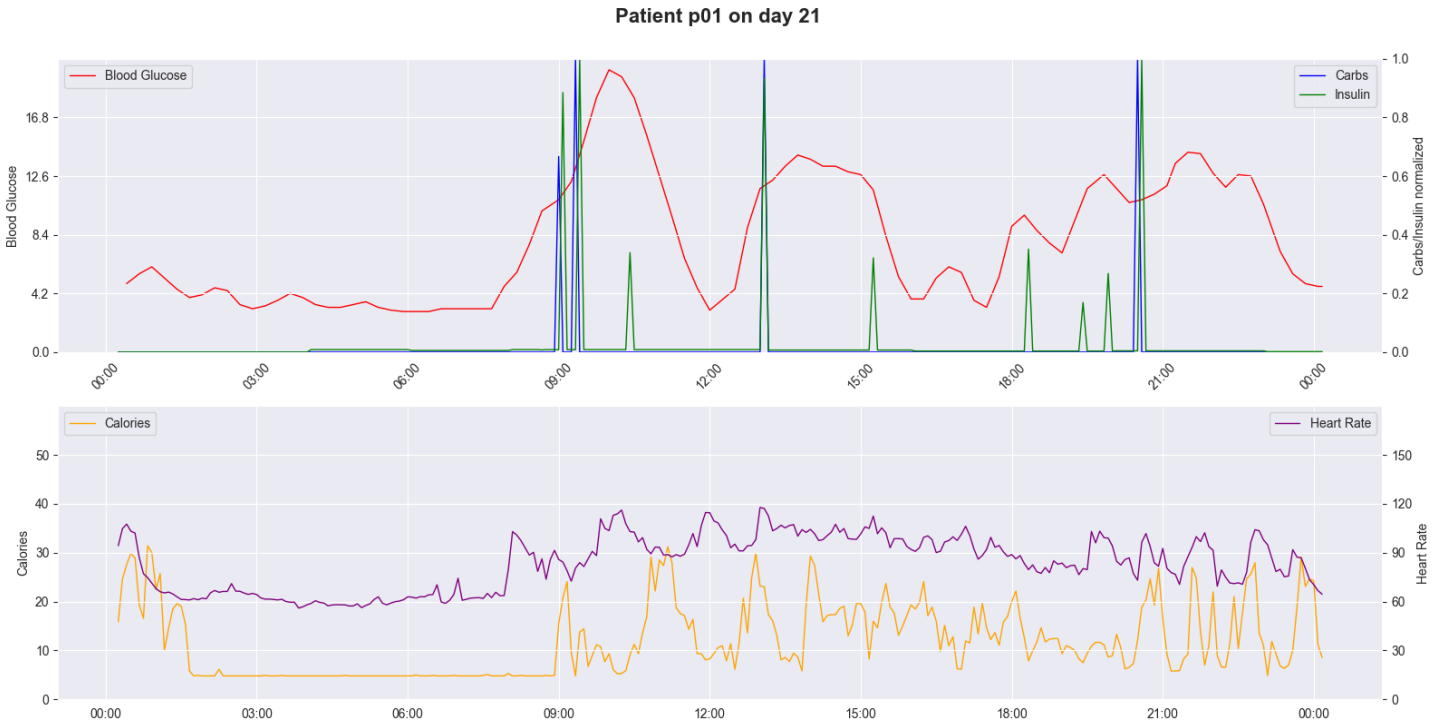
```python
def plot(patient_id: str, start_day: int, end_day: int | None = None):
    end_day = start_day if end_day is None else end_day

    patient = all_data[all_data.p_num == patient_id]
    patient_range = patient[(patient.days_since_start >= start_day) & (patient.days

    # fill missing values with linear interpolation
    patient_range_bg = patient_range['bg']
    patient_range_bg = patient_range_bg.interpolate(method='linear')

    # normalize the carbs to be between 0 and 1, fill missing values with 0
    patient_range_carbs = patient_range['carbs']
    patient_range_carbs = patient_range_carbs.fillna(0)
    patient_range_carbs = (patient_range_carbs - patient_range_carbs.min()) / (pati

    # normalize the insulin to be between 0 and 1
    patient_range_insulin = patient_range['insulin']
    patient_range_insulin = patient_range_insulin.fillna(0)
    patient_range_insulin = (patient_range_insulin - patient_range_insulin.min()) /

    patient_range_cals = patient_range['cals']
    patient_range_cals = patient_range_cals.interpolate(method='linear')

    patient_range_heart_rate = patient_range['hr']
    patient_range_heart_rate = patient_range_heart_rate.interpolate(method='linear'

    fig, [ax1, ax2] = plt.subplots(2, 1, figsize=(16, 8))
    title = f'Patient {patient_id} from day {start_day} to {end_day}' if end_day !=
    fig.suptitle(title, fontsize=16, fontweight='bold', verticalalignment='baseline

    ax1.plot(patient_range_bg.index, patient_range_bg, color='red', linestyle='-',
    ax1.set_ylabel('Blood Glucose')
    ax1.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))
    ax1.tick_params(axis='x', rotation=45)
    max_bg = np.ceil(np.max(patient_range_bg))
    ax1.set_ylim(bottom=0, top=max_bg)
    ax1.set_yticks(np.arange(0, max_bg, max_bg / 5))

    ax1_2 = ax1.twinx()
    ax1_2.plot(patient_range_carbs.index, patient_range_carbs, color='blue', linest
    ax1_2.plot(patient_range_insulin.index, patient_range_insulin, color='green', l
    ax1_2.set_ylabel('Carbs/Insulin normalized')
    ax1_2.set_ylim(bottom=0, top=1)

    # Add legends for both line and bar plots
    ax1.legend(loc='upper left')
    ax1_2.legend(loc='upper right')

    ax2.plot(patient_range_cals.index, patient_range_cals, color='orange', linestyl
    ax2.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))
    ax2.set_ylabel('Calories')
    ax2.set_ylim(bottom=0, top=60)
    ax2.set_yticks(np.arange(0, 60, 10))
```

```
    ax2_1 = ax2.twinx()
    ax2_1.plot(patient_range.index, patient_range_heart_rate, color='purple', lines
    ax2_1.set_ylabel('Heart Rate')
    ax2_1.set_ylim(bottom=0, top=180)
    ax2_1.set_yticks(np.arange(0, 180, 30))

    ax2.legend(loc='upper left')
    ax2_1.legend(loc='upper right')

    plt.xlabel('Time')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
```

Example: Patient 1 - Day 21

This example illustrates a typical day for patient `p01` by visualizing various parameters over the course of Day 21. Below is the plotted visualization for the parameters blood glucose levels, the carbs and insulin intake, the calories burned and the heart rate, providing insights into the relationships and dynamics of these variables throughout the day.

```
plot('p01', 21)
```



The plots reveal significant blood glucose fluctuations throughout the day, with noticeable spikes around potential meal times. Carbs and insulin intake align closely with these spikes, indicating the efforts to manage post-meal glucose levels. Insulin doses are applied in tandem with carb intake, which follows typical diabetes management practices.

Physical activity is reflected in increased calories burned and elevated heart rate, particularly in the morning. This activity likely contributes to overall blood glucose stability and highlights its role in glucose management.

Data Distribution

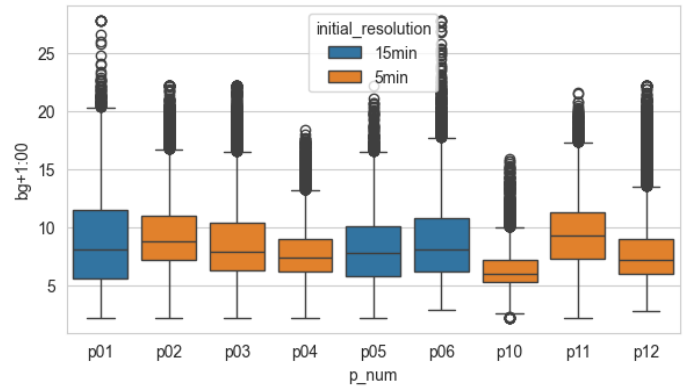
In this section, we present visualizations of the distributions for both the independent variables (features) and the target variable. These visualizations provide insights into the range, central tendencies, and variability of the dataset, helping to understand the underlying patterns and relationships within the data.

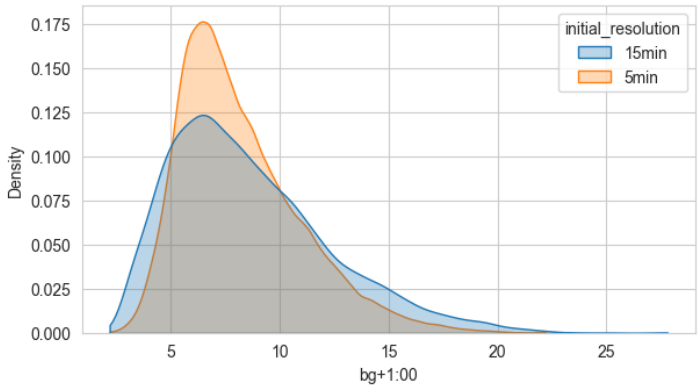
```
# import requiered libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Hide warnings
import warnings
warnings.filterwarnings('ignore')
```

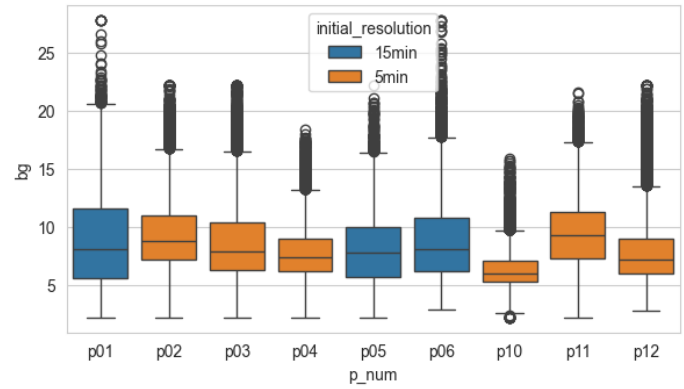
```
# Read the 'all_train.csv' file, to get a table of time resolution
all_train = pd.read_csv( '../../../data/interim/all_train.csv' ,  parse_dates = [0]
all_train.head()

# NOTE: some entries in activity are really the same category. For example 'Walk' &
# Here, we convert those cases to the same category
conv_d = {
    'Walking':'Walk',
    'Running':'Run',
    'Weights':'Strength training'

}

all_train['activity'] = all_train['activity'].replace( conv_d )
```

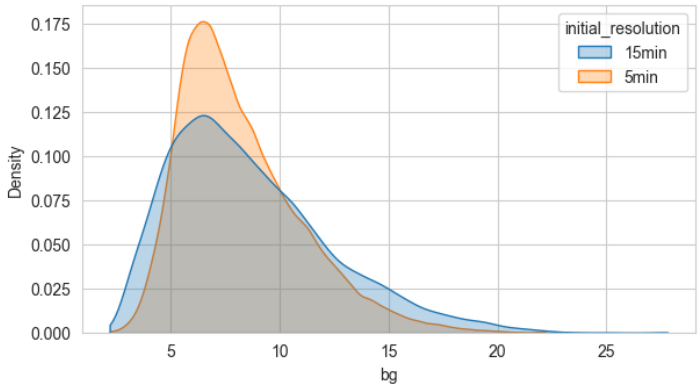
Distribution of all Features and the Target Variable

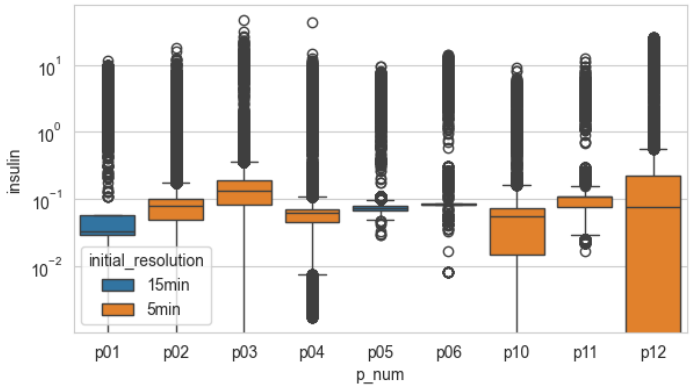
```
metrics_l = ['bg+1:00','bg','insulin','carbs','hr','steps','cals']

fig,ax = plt.subplots(7,2,figsize=(15,30))
ax_ix = 0

for metric in metrics_l:
    # print( metric )
    sns.boxplot(
        data = all_train.reset_index() ,
        x = 'p_num' ,
        y = metric ,
        hue = 'initial_resolution' ,
        ax = ax[ ax_ix , 0 ]
        )

    sns.kdeplot(
        data = all_train.reset_index()  ,
        x = metric ,
        hue = 'initial_resolution' ,
        ax = ax[ ax_ix , 1 ] ,
        common_norm = False ,
        fill = True ,
        alpha = .3 ,
        cut = 0
        )

    if metric not in ['bg+1:00','bg','hr']:
        ax[ ax_ix , 0 ].set_yscale( 'log' )
        ax[ ax_ix , 1 ].set_xscale( 'log' )

    ax_ix += 1
```

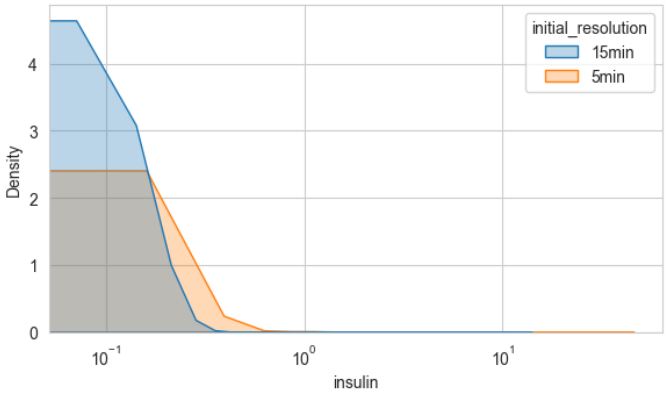

initial_resolution
15min
5min
bg+1:00
p_num
p01
p02
p03
p04
p05
p06
p10
p11
p12

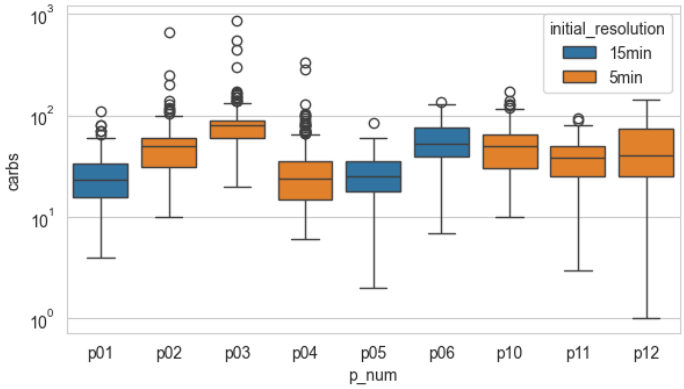

initial_resolution
15min
5min
Density
bg+1:00

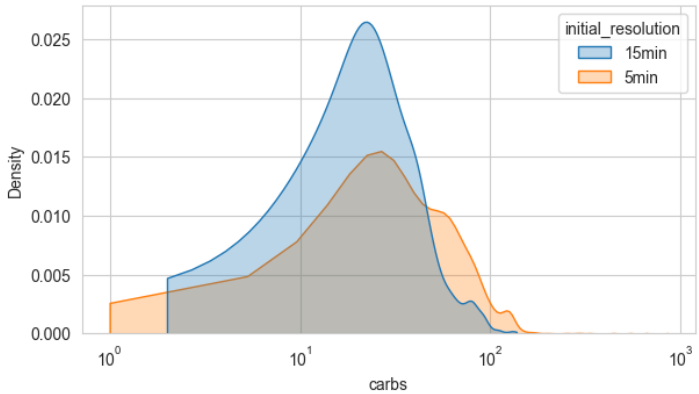

initial_resolution
15min
5min
bg
p_num
p01
p02
p03
p04
p05
p06
p10
p11
p12

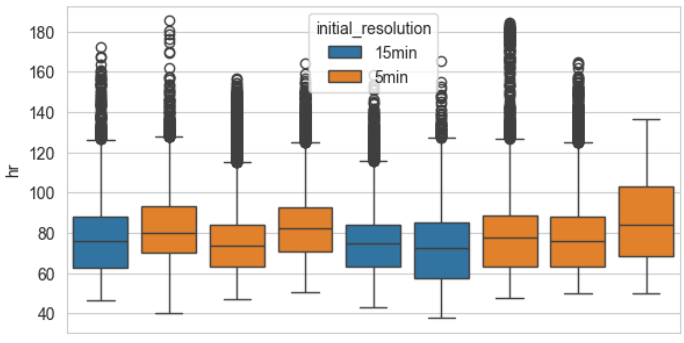

initial_resolution
15min
5min
Density
bg

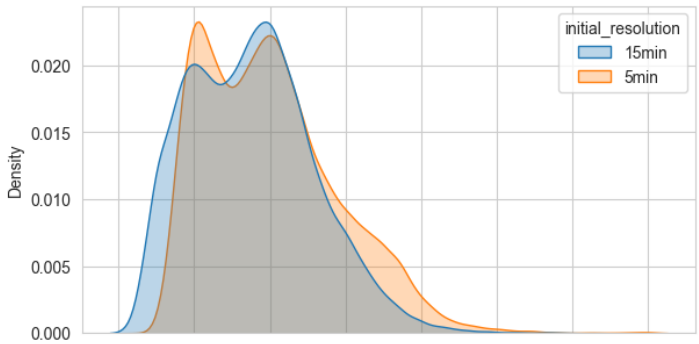

initial_resolution
15min
5min
insulin
p_num
p01
p02
p03
p04
p05
p06
p10
p11
p12

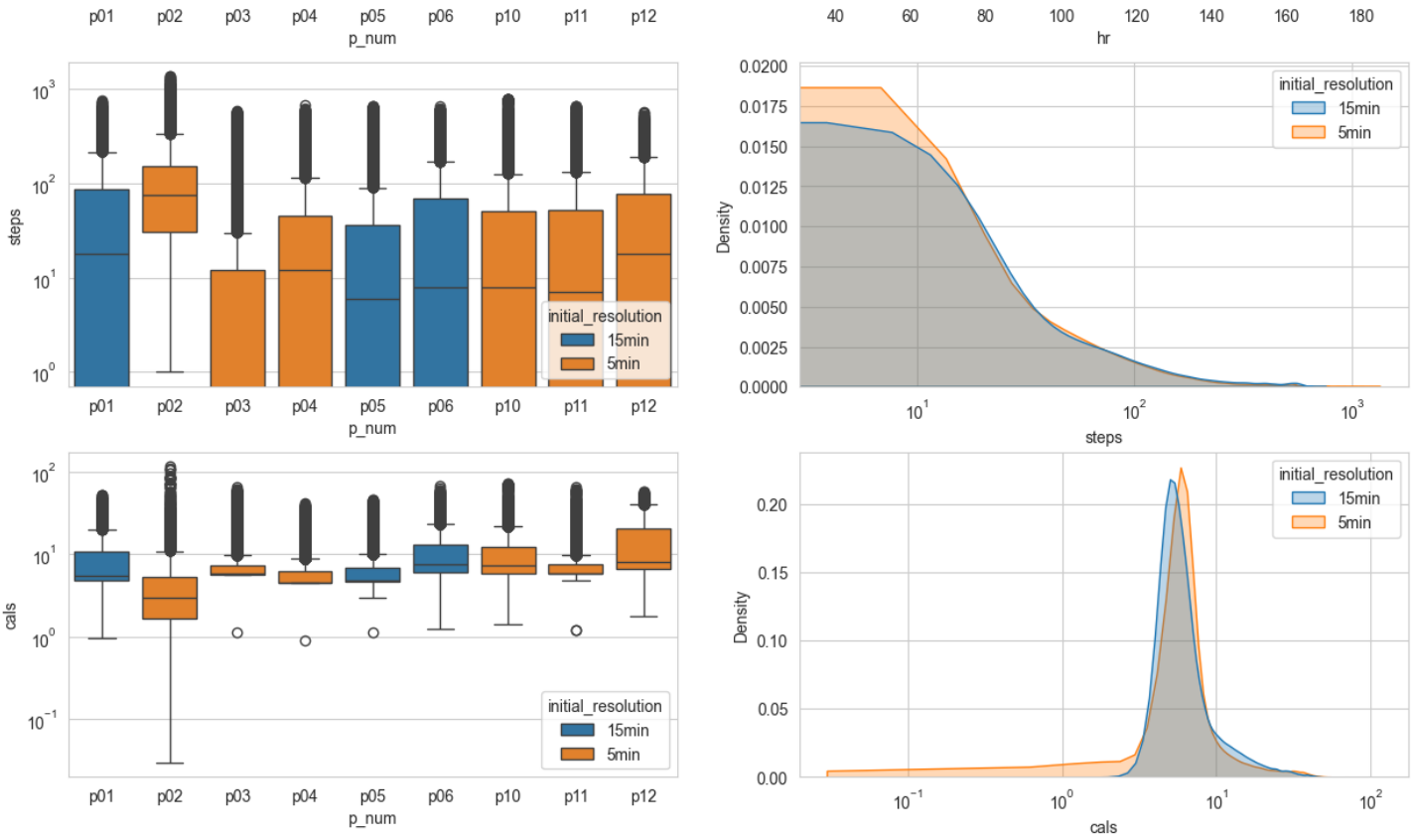

initial_resolution
15min
5min
Density
insulin


initial_resolution
15min
5min
carbs
p_num
p01
p02
p03
p04
p05
p06
p10
p11
p12


initial_resolution
15min
5min
Density
carbs


initial_resolution
15min
5min
hr
p_num
p01
p02
p03
p04
p05
p06
p10
p11
p12

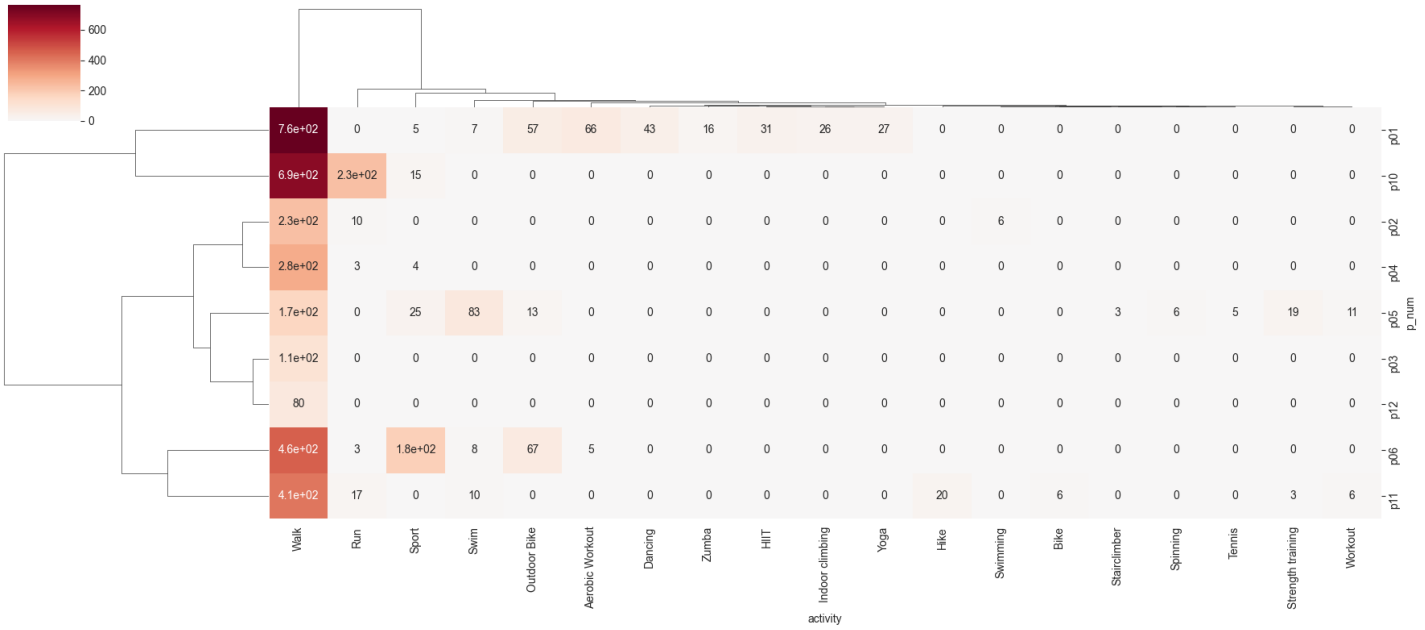

initial_resolution
15min
5min
Density



Global Activity Levels for Each Patient

The plot below illustrates the global activity levels across the entire dataset for each patient. It counts the number of times specific activities are logged for every patient, offering insights into individual activity patterns.

```
activity_df = pd.crosstab( all_train.p_num , all_train.activity )
sns.clustermap( activity_df , center = 0 , cmap = "RdBu_r" , annot = True , figsize
```



Summary

Observations on Distributions

- Blood Glucose Levels (`bg` and `bg+1:00`):
  - There is variability across patients, but most distributions have a similar central tendency. Some patients, like `p12`, show more outliers and higher ranges, indicating irregular blood glucose patterns.
  - Both distributions are positively skewed, with more values clustered in lower ranges.
- Insulin and Carbs:
  - Both show right-skewed distributions with occasional outliers. For instance, patient `p03` shows higher carbohydrate consumption, which is reflected in increased insulin usage.
- Heart Rate (hr):
  - The heart rate distributions are relatively consistent across patients, with slight variations in the median and range.
  - It follows a roughly normal distribution but is slightly right-skewed, with most values clustering between 60–100 beats per minute.
- Steps:
  - Steps show a large variability across patients, with some (e.g., `p03` and `p05`) displaying higher median activity levels and wider interquartile ranges.

    - The data for both 5 min (orange) and 15 min (blue) resolutions are positively skewed, with most values clustering at lower ranges. The 5 min resolution shows finer granularity, with a smoother distribution.
- Calories Burned (cals):
    - The calorie distributions are more consistent across patients, with fewer outliers compared to steps. Patients like `p01` and `p02` show higher calorie expenditure, possibly reflecting more physical activity.
    - `cals` shows a similar trend across resolutions, with most values concentrated in lower ranges. The 5 min resolution captures more detailed variability compared to the broader trends seen in the 15 min resolution.
- Initial Resolution Comparison (5 min vs. 15 min): The 5 min resolution (orange) results in smoother and denser distributions, reflecting finer granularity in data collection. The 15 min resolution (blue) captures broader patterns but with less precision.

Activity Distribution Across Patients

- The heatmap shows that the activities logged vary significantly among patients.
- Certain activities, such as “Walk,” are common across multiple patients and have the highest logging frequency overall.
- Patients demonstrate distinct activity patterns. For instance, Patient `p01` shows high activity across various activities, while Patient `p10` is particularly active in running.

Conclusion:

The analyzed features, including activity levels, blood glucose, insulin, and carbohydrate intake, show significant variability and skewed distributions across patients, with occasional spikes indicating irregularities of activity. Activity patterns are highly individualized, with some patients engaging in diverse activities while others specialize in specific ones. These differences, combined with finer granularity provided by the 5 min resolution, highlights the need for personalized analysis of the complex interplay between physical activity and glucose management.

Data Correlation

In this section, we analyze and visualize the relationships between the independent variables and the target variable. By examining correlations, we aim to identify how strongly each feature

influences the target variable, gaining insights into the feature selection process for modelling.

```
# import requiered libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Hide warnings
import warnings
warnings.filterwarnings('ignore')
```

```
# Read raw train data
train = pd.read_csv( '../../../data/raw/train.csv' , index_col = 0 )
train.head()
```

	p_num	time	bg-5:55	bg-5:50	bg-5:45	bg-5:40	bg-5:35	bg-5:30	bg-5:25	bg-5:20	...	acti
id												
p01_0	p01	06:10:00	NaN	NaN	9.6	NaN	NaN	9.7	NaN	NaN	...	
p01_1	p01	06:25:00	NaN	NaN	9.7	NaN	NaN	9.2	NaN	NaN	...	
p01_2	p01	06:40:00	NaN	NaN	9.2	NaN	NaN	8.7	NaN	NaN	...	
p01_3	p01	06:55:00	NaN	NaN	8.7	NaN	NaN	8.4	NaN	NaN	...	
p01_4	p01	07:10:00	NaN	NaN	8.4	NaN	NaN	8.1	NaN	NaN	...	

5 rows × 507 columns

```
# Read the 'all_train.csv' file, to get a table of time resolution
all_train = pd.read_csv( '../../../data/interim/all_train.csv' ,  parse_dates = [0]
display( all_train.head() )

time_res = all_train[['p_num','initial_resolution']].drop_duplicates()
time_res_d = dict( zip( time_res.p_num , time_res.initial_resolution ) )
# pprint( time_res_d )

#Add the time resolution to the raw data
train['initial_resolution'] = train['p_num'].map( time_res_d )

# NOTE: some entries in activity are really the same category. For example 'Walk' &
# Here, we convert those cases to the same category
conv_d = {
    'Walking':'Walk',
    'Running':'Run',
    'Weights':'Strength training'

}

all_train['activity'] = all_train['activity'].replace( conv_d )
```

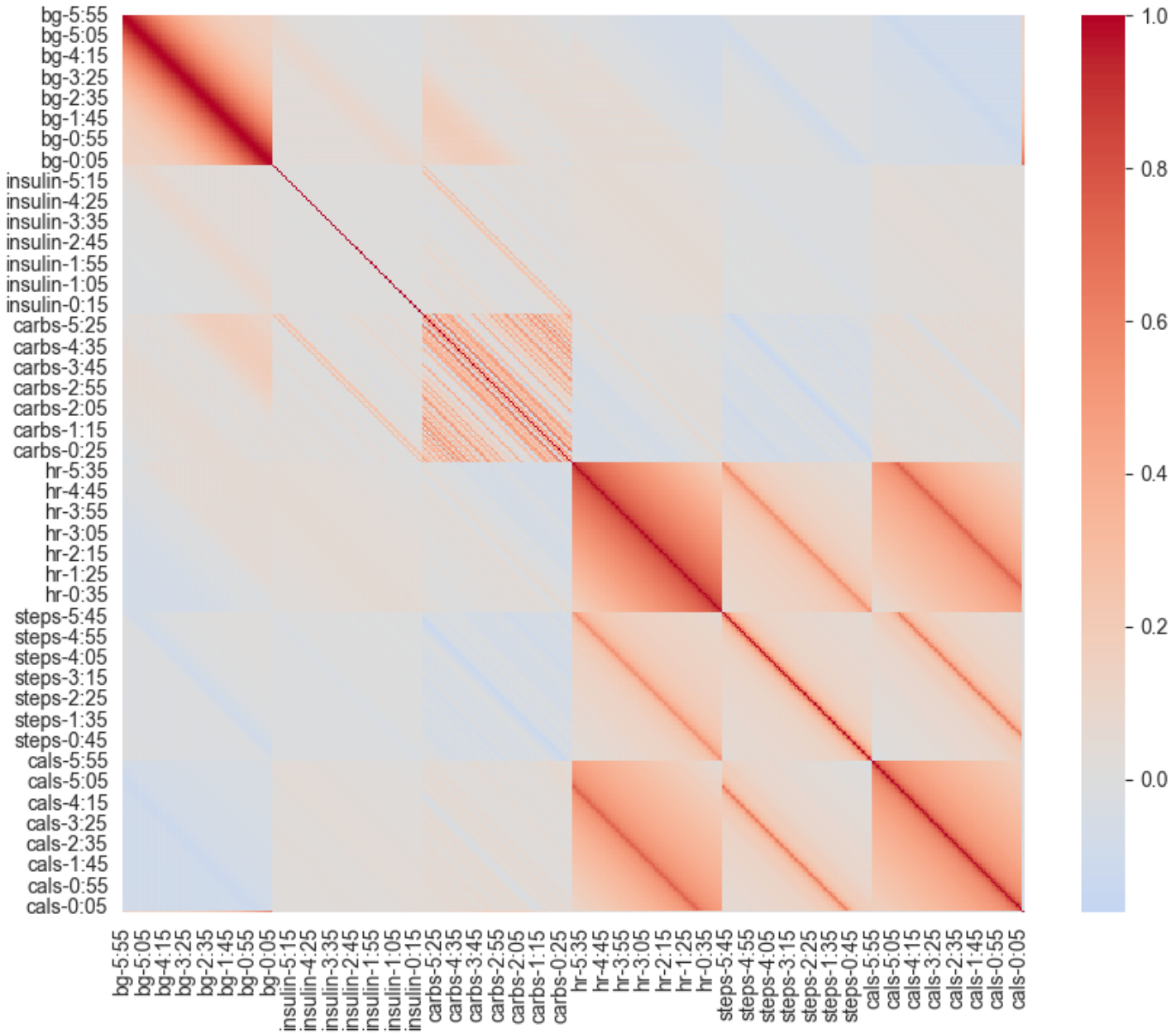
	p_num	days_since_start	time	initial_resolution	bg	insulin	carbs
2020-01-01 00:15:00	p01	0	00:15:00	15min	NaN	0.0083	NaN
2020-01-01 00:20:00	p01	0	00:20:00	15min	NaN	0.0083	NaN
2020-01-01 00:25:00	p01	0	00:25:00	15min	9.6	0.0083	NaN
2020-01-01 00:30:00	p01	0	00:30:00	15min	NaN	0.0083	NaN
2020-01-01 00:35:00	p01	0	00:35:00	15min	NaN	0.0083	NaN

Global Correlations Between Lag Features and Target Variable

To understand the relationships between the lag features and the target variable, we compute and visualize their correlations within the raw dataset. The heatmap below highlights these global

correlations, providing insights into how features interact and influence the target variable.

```
plt.figure( figsize=(10,8) )
corr_df = train.select_dtypes( include = np.number ).corr()
sns.heatmap( corr_df , center = 0 , cmap = "coolwarm" );
```



Observations from the Heatmap

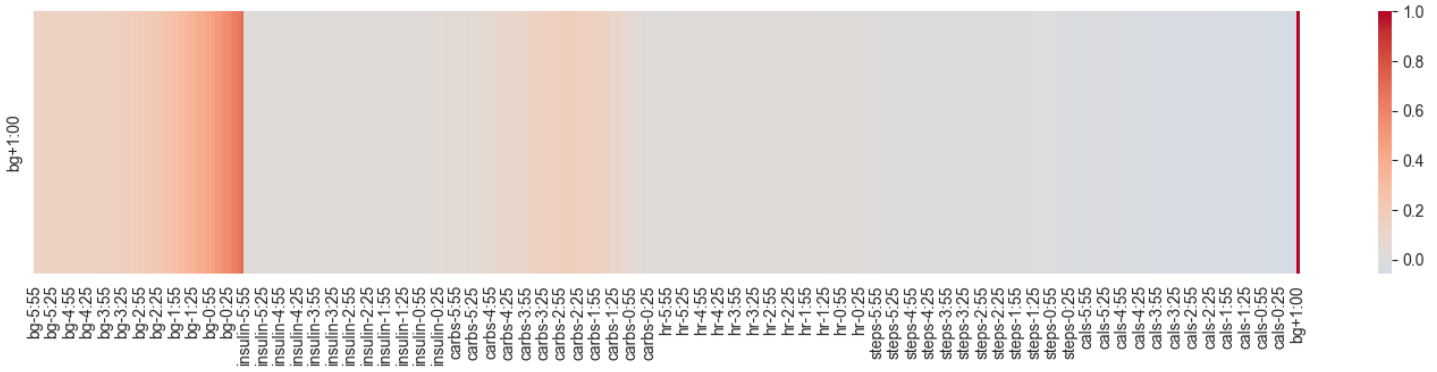
The heatmap highlights the temporal consistency within each variable and weaker inter-variable relationships. Strong self-correlations across time lags indicate temporal consistency in the data, where previous values closely predict subsequent ones. The weak or moderate correlations

between different features suggest more complex, indirect dependencies that may require advanced modelling to predict effectively.

Highlight of the Correlation of the Target Variable Against All Features

To focus specifically on how `bg+1:00` correlates with all other features in the raw dataset, we visualize its correlations in the heatmap below. This helps identify which features have the strongest influence on predicting future blood glucose levels.

```
plt.figure( figsize=(18,3) )
sns.heatmap( pd.DataFrame( corr_df.loc[:,'bg+1:00'] ).T , center = 0 , cmap = "cool
```



The heatmap highlights that `bg+1:00` is most strongly correlated with recent lagged bg values, reflecting the temporal dependency of blood glucose levels. Moderate correlations are observed with features like insulin and carbs, while activity-related features (steps, hr, cals) show weaker correlations, indicating their more indirect impact on future blood glucose.

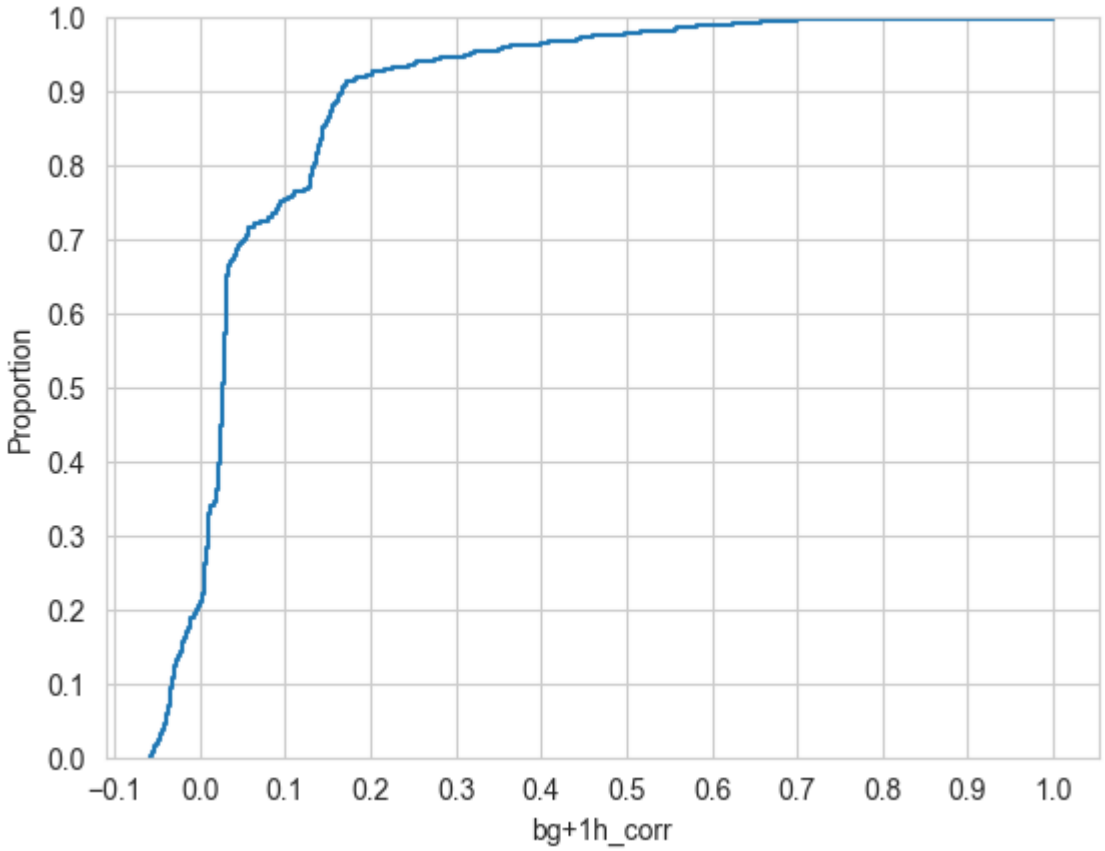
Cumulative Distribution of Lag Feature Correlations to `bg+1:00`

To analyze how strongly different lagged features correlate with the target variable `bg+1:00`, we calculate the Pearson correlation coefficients and plot their cumulative distribution function (ECDF). This visualization highlights the distribution of correlation strengths across all lagged features.

1. Correlation Extraction:
    - Pearson correlation coefficients for all lagged features with `bg+1:00` are computed and grouped by feature type (e.g., `bg`, `insulin`, `carbs`, etc.).
    - The average correlation for each feature type is calculated.

2. ECDF Plot:
    - The ECDF shows the cumulative proportion of features that achieve a given level of correlation with `bg+1:00`.

```
# Extract the pearson r coefficients
bg_plus_1_corr = pd.DataFrame( corr_df.loc[:,'bg+1:00'] ).sort_values( 'bg+1:00' ,
bg_plus_1_corr.columns = ['column','bg+1h_corr']
bg_plus_1_corr[['metric','time']] = bg_plus_1_corr['column'].str.split('-',expand=T
bg_plus_1_corr.groupby('metric').agg({'bg+1h_corr':np.mean})

# Plot the ECDF
sns.ecdfplot(data=bg_plus_1_corr, x="bg+1h_corr")
plt.xticks( np.arange(-0.1,1.01,.1) )
plt.yticks( np.arange(0,1.01,.1) );
```



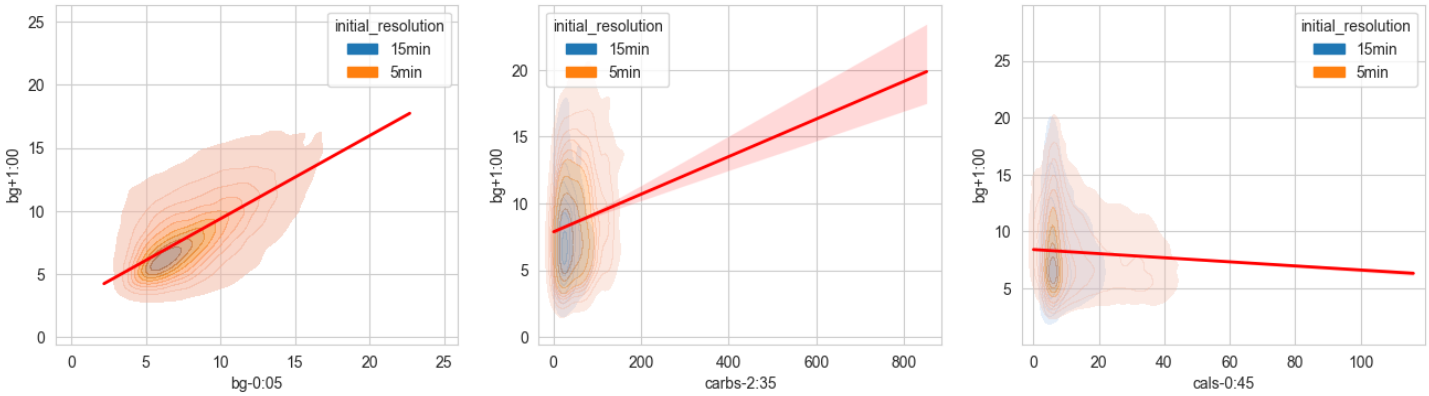
Observations

The ECDF shows that lagged `bg` features have the strongest correlations with `bg+1:00`, dominating the higher correlation range. In contrast, features like insulin, carbs, steps, hr, and cals exhibit weaker correlations, contributing less to the prediction of future glucose levels.

Examples of High-, Mid- and Low-Correlataed Features with `bg+1:00`

To demonstrate the relationship between features and the target variable `bg+1:00`, we visualize the density of data points and overlay a regression line (red) of least squares.

```
fig,ax = plt.subplots(1,3,figsize=(16,4))
ax = ax.flatten()

sns.kdeplot( data = train , y = 'bg+1:00' , x = 'bg-0:05' , hue = 'initial_resoluti
sns.regplot( data = train , y = 'bg+1:00' , x = 'bg-0:05' , scatter = False , color

sns.kdeplot( data = train , y = 'bg+1:00' , x = 'carbs-2:35' , hue = 'initial_resol
sns.regplot( data = train , y = 'bg+1:00' , x = 'carbs-2:35' , scatter = False , co

sns.kdeplot( data = train , y = 'bg+1:00' , x = 'cals-0:45' , hue = 'initial_resolu
sns.regplot( data = train , y = 'bg+1:00' , x = 'cals-0:45' , scatter = False , col
```



Observations

1. High-correlated feature (`bg-0:05`):
    - The plot shows a strong positive linear relationship with the target variable, as indicated by the clear density alignment along the regression line. This highlights the predictive importance of recent blood glucose levels.
2. Mid-correlated feature (`carbs-2:35`):
    - The plot shows a moderate relationship to `bg+1:00`, with some density clustering but less alignment along the regression line compared to `bg-0:05`. This suggests a weaker but still relevant influence of carbohydrate intake on future blood glucose levels.
3. Low-correlated feature (`cals-0:45`):

- The plot exhibits a minimal relationship to `bg+1:00`, with widely dispersed density and poor alignment with the regression line. This indicates that calorie expenditure has little direct influence on predicting future blood glucose levels.

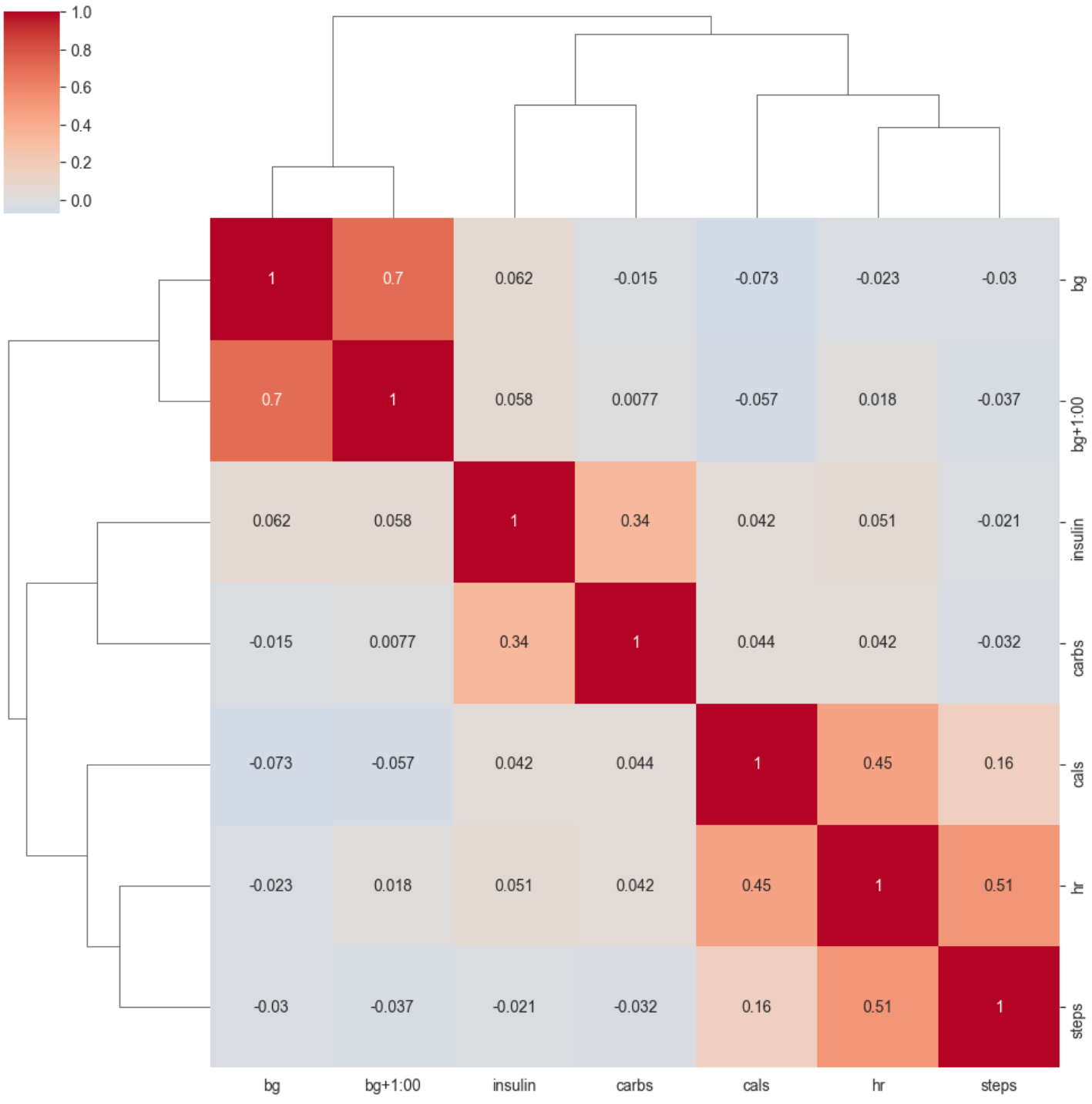
These examples emphasize the varying degrees of feature relevance, with `bg` lag features being the most predictive and activity-related features contributing less.

Correlation Between All Numeric Variables as a Time Series

This visualization shows the correlation between all numeric variables (bg, insulin, carbs, hr, steps, cals, and bg+1:00) using a clustered heatmap. By organizing features based on their correlation patterns, this plot highlights relationships and clusters of interdependent variables.

```
plt.figure( figsize=(3,3) )
sns.clustermap(
    all_train[['bg','insulin','carbs','hr','steps','cals','bg+1:00']].corr() ,
    center = 0 ,
    cmap = "coolwarm" ,
    annot = True
    );
```

```
<Figure size 300x300 with 0 Axes>
```



Conclusion

This clustered heatmap reveals once again clear groupings of variables, emphasizing the dominant role of lagged `bg` and related features in predicting future glucose levels, while activity metrics show weaker relevance.

Summary

- Lagged `bg`-features exhibit the strongest correlation with `bg+1:00`, underscoring the temporal dependency of blood glucose levels.
- Features such as `carbs` and `insulin` show moderate correlations with `bg+1:00`, reflecting their role in glucose regulation.
- Activity-related metrics (`steps`, `cals`, `hr`) have weak correlations with `bg+1:00`, indicating their indirect or limited impact on predicting future glucose levels.
- A notable correlation exists between `carbs` and `insulin`, highlighting their relationship in dietary management.

Exploratory Data Analysis Conclusions

- Anomalies:
  - A minor anomaly was detected for `p12`, where two negativ values were identified as irregular.
  - Only one patient (`p11`) shows small inconsistencies in continuous rows within the raw data.
- Missing Data:
  - Feature columns in the raw data generally have moderate fractions of missing values (NAs).
  - The columns related to `activity` and `carbs` have a substantial missing values with over 95%.
- Feature Distributions:
  - With only minor exceptions, the distribution of features and target variable is largely comparable across patients.
  - All variables exhibit right-skewed distributions.
- Correlations:
  - Most features, whether considered as whole time series or lagged features, show weak correlations with `bg+1:00`, emphasizing the predictive dominance of recent `bg` values.

Business Relevant Insights

- **Data challenges:** The dataset provides valuable variables known to influence blood glucose levels, but using some of them directly is challenging. For instance, columns related to activity and carbohydrates have over 95% missing values, making them difficult to include in modelling without substantial data imputation. Additionally, the activity data includes multiple activity types, requiring specialized encoding methods to make the data usable.
- **Predictive power of prior blood glucose:** Prior blood glucose (`bg`) levels are strongly correlated with future glucose (`bg+1:00`), while other variables show weak correlations. This suggests that a well-performing model could be built using only prior glucose levels, potentially reducing the need for additional data or complex devices. However, integrating other variables could still enhance the model's performance if the challenges related to their data quality and usability are resolved.

This analysis underscores the importance of balancing data complexity with practicality in designing products or technologies for blood glucose prediction.

Planned Steps for Upcomming Analysis

Preprocessing & Preliminary Modelling:

1. Raw data preprocessing
    - Impute the raw data using the column median (to address skewed distributions) and run LazyPredict.
2. Processed data preprocessing
    - Impute the processed data ("long format") using either the column median or time series interpolation and run LazyPredict.

Note/Reference

(Lazy Predict)[shankarpandala/lazypredict] is a Python library designed to build a variety of basic models for both classification and regression tasks with minimal coding effort. It allows for a quick comparison of models to understand which ones perform better on a given dataset, all without requiring parameter tuning. This makes it a valuable tool for exploratory modelling and baseline performance evaluation.

Preprocessing

The objective of this section is to show and to explain the various preprocessing steps and feature engineering techniques that have been applied by our team in the context of the Kaggle competition. These steps are essential to transform the raw data into a format suitable for modelling and to maximize the performance of predictive models.

Preprocessing steps

This section outlines the key preprocessing steps, carried out by our team, which, in our opinion, were necessary to properly prepare the dataset for different types of modelling.

We developed a bunch of custom transformers tailored to the structure and characteristics of our dataset. These transformers are used to preprocess the data both during training and prediction phases, ensuring consistency and efficiency.

- Objectives of custom transformers:
  - Handle missing values and anomalies.
  - Format the data to align with the requirements of time series.
  - Generate additional features that might improve the model's predictive performance.

Preprocessing Steps

In this section, we outline the pre-processing steps that in our opinion are required to format the data properly for different modelling approaches.

Transformers

Following custom transformers were developed that are designed to handle the unique characteristics of our dataset. They can be then used in preprocessing pipelines for both training and prediction phases.

- `DataTimeHourTransformer`: Encodes time-of-day information as either sine and cosine values or discrete bins.

- `DateTimeTransformer`: Converts the timestamp column in a DateTime format and optionally renames the column allowing the original column to be dropped.
- `DayPhaseTransformer`: Categorizes timestamps into day phases (e.g., morning, noon, night) based on predefined ranges.
- `DropColumnsTransformer`: Removes specified columns or columns whose names start with given prefixes from a DataFrame (e.g. `carbs-*`).
- `ExtractColumnsTransformer`: Selects specific columns for modelling.
- `FillPropertyNaNsTransformer`: Imputes missing values (NaNs) for specific variables (e.g., `bg`, `insulin`) using various methods such as mean, median, zero, or interpolation and forward/backward fill options.
- `GetDummiesTransformer`: Converts categorical features into one-hot encoded format.
- `Log1pTransformer`: Applies a logarithmic transformation to reduce skewness in numerical features.
- `MinMaxScalerTransformer`: Scales numerical features to a defined range, typically [0, 1].
- `PropertyOutlierTransformer`: Detects outliers in columns related to a specified parameter using a user-defined filter function and replaces them using a configurable fill strategy such as zero, min, max, mean, or median.
- `RollingAverageTransformer`: Computes rolling averages for time-series features to smooth fluctuations.
- `StandardScalerTransformer`: Standardizes numerical features by removing the mean and scaling to unit variance.

For each model, these transformers can be combined flexibly into a `preprocessing pipeline` and a `scaling pipeline` to perform the data preparation to the specific needs of the model.

```
preprocessing_pipeline = Pipeline(steps=[
    ('date_time', DateTimeTransformer(source_column='time', source_time_format='%H:%
    ('drop_parameter_cols', DropColumnsTransformer(starts_with=['activity', 'carbs'
    ('drop_others', DropColumnsTransformer(columns_to_delete=['p_num', 'time'])),
    ('fill_properties_nan_bg', FillPropertyNaNsTransformer(parameter='bg', how=['in
    ('fill_properties_nan_insulin', FillPropertyNaNsTransformer(parameter='insulin'
    ('fill_properties_nan_cals', FillPropertyNaNsTransformer(parameter='cals', how=
    ('fill_properties_nan_hr', FillPropertyNaNsTransformer(parameter='hr', how=['in
    ('fill_properties_nan_steps', FillPropertyNaNsTransformer(parameter='steps', ho
    ('drop_outliers', PropertyOutlierTransformer(parameter='insulin', filter_functi
    ('extract_features', ExtractColumnsTransformer(columns_to_extract=columns_to_ex
])

standardization_pipeline = Pipeline(steps=[
    ('get_dummies', GetDummiesTransformer(columns=['hour'])),
    ('min_max_scaler', StandardScalerTransformer(columns=columns_to_extract[2:-1]))
])

pipeline = Pipeline(steps=[
    ('preprocessing', preprocessing_pipeline),
    ('standardization', standardization_pipeline)
])
```

Preprocessing Steps

Based on the insights gained from exploratory data analysis, the following preprocessing steps have been identified as the most appropriate for preparing the dataset for modelling:

1. Handling columns with high missing values:
    - Columns related to `carbs` and `activity` are removed due to their high fraction of missing values (>95%).
2. Fixing anomalies:
    - For Patient `p12`, two negative values in the insulin column will be corrected by replacing them with zero.
3. Selected features for modelling:
    - The lag features of the following numerical parameters will be retained for modelling:
        - `bg`: Blood glucose
        - `insulin`: Insulin
        - `carbs`: Carbohydrates
        - `hr`: Heart Rate
        - `steps`: Steps

        - `cals`: Calories
4. Imputation and standardization:
    - Missing values will be imputed after splitting the data into train and test sets. We'll use interpolation and medians techniques for both train and test sets.
    - Numerical columns will be standardized, for example, using StandardScaler(), to ensure consistent scaling across features.
5. Feature Engineering:
    - Additional features, such as a time of day categorical column (e.g., Morning, Afternoon, Evening), can be created from the timestamp data to provide contextual information.
    - Once a well-performing model is established, the dataset can be revisited to incorporate the removed activity and carbohydrate columns to evaluate their impact on model performance.

Modelling

In this section, we present and explain two different approaches to modelling the data. The first approach is based on the train data given by the competition, while the second one uses the test data for a creation of augmented train data.

Approaches Based on the Raw Data

We drove two different approaches to preprocess the data and to model them.

1. **Traditional Train Data Approach**

- A global model was created using only the given train data (from 8 patients) to predict blood glucose levels for all participants.
- This approach does not account for individual patient characteristics but provides a baseline model for comparison.

2. **Train and Test Data Approach**

- To address the limitations of the traditional approach, this method incorporates test data to capture individual patient characteristics.

- Synthetic train data were created from the test data by shifting lag features to the right, using previous hours of data to predict future hours. This leverages the time-series nature of the dataset to augment the training process.

Steps for Both Approaches

For both approaches, we'll perform the following steps:

- Apply relevant preprocessing steps to prepare the data.
- Run LazyPredict on a large subset of the features (up to 300 columns).
- Define the most important features using SHAP (SHapley Additive exPlanations) or other feature importance methods.
- Identify the most promising models running LazyPredict again, but now with the most important features.
- Perform hyperparameter optimization on the most promising models using SciKit-Optimize for fine-tuning.
- Combine the best-performing models into **ensembles** to enhance prediction accuracy.
- Use the final models to predict blood glucose levels on the test data.

Model 1 - Train Data Only

Overview

Our first approach uses only the data contained in `train.csv` to train and tune our models. The workflow for this approach includes:

- Testing multiple regressor models with default parameters using LazyPredict
- Feature importance based on SHAP values
- Re-running LazyPredict on the feature subset derived from the SHAP analysis
- Hyperparameter Tuning on Feature Set, Ensemble Regressor & Kaggle submissions

Usage of LazyPredict

- We are using LazyPredict nightly in the project.

- Why LazyPredict? LazyPredict (LP) is an AutoML tool designed for quick testing of multiple estimators (classifiers or regressors) from Scikit-learn. Under the hood, LP defines a list of models to test. It simplifies initial model selection by providing performance metrics for a variety of models without requiring parameter tuning.
- Challenges LazyPredict:
  In case of Regression, two models from the default list took an unusually long time to process and even caused Jupyter kernels to crash. To circumvent this, the list of regressors was modified to exclude these problematic models.
- Customizing LazyPredict: To streamline the process and customize the list of regressors tested, a helper function get_lazy_regressor() was created. This function can be imported from `LazyPredict.py` in the `src/helpers` folder. It provides a convenient instantiation of `LazyRegressor()` with a tailored list of regressors.

Feature Importance via SHAP (SHapley Additive exPlanations) Analysis

- SHAP (SHapley Additive exPlanations) is a framework for interpreting machine learning model predictions by attributing the influence of each input feature to the output. SHAP analysis is especially valuable in improving model transparency, identifying key features, and building trust in machine learning applications. It is widely used in critical fields like healthcare, finance, and regulatory environments where interpretability is essential.
- **Key Concepts**:
  - **Baseline Prediction**: The average prediction when no features are included.
  - **Feature Contribution**: Each SHAP value represents a feature's contribution to moving the model's prediction away from the baseline.
- **Advantages**:
  - **Model-Agnostic**: Can be applied to any machine learning model.
  - **Fair Attribution**: Ensures features are credited proportionally to their impact.
  - **Local and Global Interpretability**: Explains individual predictions and overall model behavior.

Multiple Regressor Models With Default Parameters Using LazyPredict

After preprocessing the data using our custom pipeline, we used LazyPredict to evaluate multiple regression models.

```
# import get_lazy_regresor()
import importlib.util
import sys

def import_function_from_path(file_path, function_name):
    # Load the module from the file path
    spec = importlib.util.spec_from_file_location("module_name", file_path)
    module = importlib.util.module_from_spec(spec)
    sys.modules["module_name"] = module
    spec.loader.exec_module(module)

    # Retrieve the function from the loaded module
    func = getattr(module, function_name)
    return func

# Below, PATH_TO_PROJECT is the location of the project folder (e.g: /home/sep24_bd
PATH_TO_SCRIPT = 'PATH_TO_PROJECT/notebooks/helpers/LazyPredict.py'
function_name = 'get_lazy_regressor'

get_lazy_regressor = import_function_from_path( PATH_TO_SCRIPT , function_name )

PATH_TO_SRC = '/Users/masaver/Desktop/masaver/data_science_projects/sep24_bds_int_m
sys.path.append( PATH_TO_SRC )

# Load other requiered libraries
import os
import pandas as pd
from sklearn.model_selection import train_test_split

# Mute warnings
import warnings
warnings.filterwarnings("ignore")

# Import the preprocessign pipeline
from pipelines import *
```

```
# Read the data from train.csv
data_dir = '../../../../data/'
train_file = os.path.join( data_dir , 'raw' , 'train.csv' )
df_train = pd.read_csv(train_file, index_col = 0 , parse_dates = True )
display( df_train.head() )
```

	p_num	time	bg-5:55	bg-5:50	bg-5:45	bg-5:40	bg-5:35	bg-5:30	bg-5:25	bg-5:20	...	acti
id												
p01_0	p01	06:10:00	NaN	NaN	9.60	NaN	NaN	9.70	NaN	NaN	...	
p01_1	p01	06:25:00	NaN	NaN	9.70	NaN	NaN	9.20	NaN	NaN	...	
p01_2	p01	06:40:00	NaN	NaN	9.20	NaN	NaN	8.70	NaN	NaN	...	
p01_3	p01	06:55:00	NaN	NaN	8.70	NaN	NaN	8.40	NaN	NaN	...	
p01_4	p01	07:10:00	NaN	NaN	8.40	NaN	NaN	8.10	NaN	NaN	...	

5 rows × 507 columns

Data Preprocessing

Main steps

- Re-encoding the `timestamp` into a `day-phase`
- Dropping the following columns: `activity`, `carbs`, `steps` , `p_num` and `time`
- Imputing NANs in the remaining columns with interpolation and medians
- Two negative values in the `insulin` column replaced with `0`
- The column `day-phase` is re-encoded using `pd.get_dummies()`
- Finally, all columns were transformed using `StandardScaler`.

```
# Split the data into Features and Target variables,
# and Standarize the features with the preprocessing pipelines
X = df_train.drop( 'bg+1:00' , axis = 1 )
y = df_train['bg+1:00']

# Train Test Split
x_train,x_test,y_train,y_test = train_test_split( X , y , test_size=0.2 , random_st
data_pipe = pipeline_s
x_train_s = data_pipe.fit_transform( x_train )
x_test_s = data_pipe.transform( x_test )

display( x_train_s )
display( x_test_s )
```

	bg-5:55	bg-5:50	bg-5:45	bg-5:40	bg-5:35	bg-5:30	bg-5:25	bg-5:20	bg-5:15	bg-5:10	...
id											
p11_1931	1.07	1.07	1.04	1.07	1.01	0.91	0.81	0.74	0.81	0.84	...
p10_9422	-0.89	-0.89	-0.82	-0.79	-0.69	-0.73	-0.79	-0.79	-0.79	-0.89	...
p06_8273	0.07	0.07	0.07	-0.03	-0.13	-0.23	-0.31	-0.40	-0.49	-0.55	...
p12_17133	-0.46	-0.49	-0.46	-0.46	-0.46	-0.49	-0.49	-0.49	-0.46	-0.46	...
p03_2346	-0.66	-0.66	-0.63	-0.66	-0.63	-0.73	-0.76	-0.76	-0.76	-0.79	...
...	...	...	...	...	...	...	...	...	...	...	...
p02_17172	-0.16	-0.16	-0.16	-0.16	-0.26	-0.33	-0.39	-0.43	-0.46	-0.49	...
p10_23964	0.84	0.90	0.97	0.97	0.91	0.84	0.74	0.54	0.37	0.24	...
p03_7966	1.90	1.87	1.80	1.67	1.57	1.41	1.24	1.14	1.01	0.87	...
p03_628	-0.43	-0.46	-0.49	-0.49	-0.49	-0.49	-0.46	-0.46	-0.43	-0.46	...
p04_4394	-0.53	-0.53	-0.56	-0.43	-0.19	-0.23	-0.23	-0.29	-0.16	-0.03	...

141619 rows × 288 columns

	bg-5:55	bg-5:50	bg-5:45	bg-5:40	bg-5:35	bg-5:30	bg-5:25	bg-5:20	bg-5:15	bg-5:10	...
id											
p11_17351	0.80	0.80	0.67	0.71	0.84	1.01	1.17	1.47	1.54	1.54	...
p10_15385	-0.82	-0.79	-0.82	-0.79	-0.83	-0.79	-0.79	-0.79	-0.69	-0.66	...
p06_2496	-1.09	-1.09	-1.09	-1.07	-1.05	-1.02	-1.03	-1.02	-1.02	-0.97	...
p10_8410	-0.63	-0.59	-0.59	-0.56	-0.53	-0.53	-0.56	-0.63	-0.72	-0.83	...
p02_2562	0.44	0.41	0.34	0.31	0.31	0.27	0.24	0.24	0.21	0.21	...
...	...	...	...	...	...	...	...	...	...	...	...
p02_24761	0.90	0.34	0.01	-0.16	-0.33	-0.59	-0.79	-0.92	-0.96	-0.99	...
p10_22597	-0.69	-0.86	-0.99	-1.06	-0.86	-0.82	-0.76	-0.96	-0.96	-0.96	...
p03_2231	0.67	0.41	0.57	0.64	0.61	0.81	1.01	0.91	0.71	0.67	...
p12_4088	-0.92	-0.96	-0.99	-0.96	-0.93	-0.92	-0.89	-0.86	-0.86	-0.79	...
p04_19811	0.34	0.24	0.07	0.01	0.01	0.01	0.17	0.41	0.41	0.54	...

35405 rows × 288 columns

Descriptive Statistics

```
# Display descriptive statistics
display( x_train_s.describe() )
display( x_test_s.describe() )
```

	bg-5:55	bg-5:50	bg-5:45	bg-5:40	bg-5:35	bg-5:30	bg-5:25	b
count	141619.00	141619.00	141619.00	141619.00	141619.00	141619.00	141619.00	14
mean	-0.00	0.00	-0.00	-0.00	0.00	-0.00	-0.00	
std	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
min	-2.02	-2.02	-2.02	-2.02	-2.02	-2.02	-2.02	
25%	-0.72	-0.72	-0.73	-0.73	-0.73	-0.73	-0.73	
50%	-0.19	-0.19	-0.19	-0.19	-0.19	-0.19	-0.19	
75%	0.54	0.54	0.54	0.54	0.54	0.54	0.54	
max	6.49	6.49	6.50	6.50	6.50	6.50	6.50	

8 rows × 288 columns

	bg-5:55	bg-5:50	bg-5:45	bg-5:40	bg-5:35	bg-5:30	bg-5:25	bg-
count	35405.00	35405.00	35405.00	35405.00	35405.00	35405.00	35405.00	354
mean	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
std	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
min	-2.02	-2.02	-2.02	-2.02	-2.02	-2.02	-2.02	
25%	-0.72	-0.72	-0.73	-0.73	-0.73	-0.73	-0.73	
50%	-0.19	-0.19	-0.19	-0.19	-0.19	-0.19	-0.19	
75%	0.54	0.54	0.54	0.54	0.54	0.54	0.54	
max	6.49	6.49	6.50	6.50	6.50	6.50	6.50	

8 rows × 288 columns

Run LazyPredict Regressor

```
# Run a Lazy Regressor
# NOTE: Preliminary test showed that SVR and Quantile Regressor are not the best pe
# So we exclude them from the LazyRegressor Task
reg = get_lazy_regressor( exclude = ['SVR','QuantileRegressor'] )
models, predictions = reg.fit( x_train_s , x_test_s , y_train , y_test )
```

```
 97%|█████████▋| 36/37 [2:26:21<00:07,  7.81s/it]
```

```
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing wa
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 73353
[LightGBM] [Info] Number of data points in the train set: 141619, number of used fea
[LightGBM] [Info] Start training from score 8.273489
```

```
100%|██████████| 37/37 [2:26:29<00:00, 237.56s/it]
```

```
models
```

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
ExtraTreesRegressor	0.70	0.70	1.63	4902.69
BaggingRegressor	0.65	0.65	1.77	218.93
XGBRegressor	0.63	0.63	1.82	5.80
HistGradientBoostingRegressor	0.60	0.60	1.90	9.92
LGBMRegressor	0.60	0.60	1.90	7.96
GradientBoostingRegressor	0.55	0.55	2.00	2459.81
LassoLarsCV	0.52	0.53	2.06	7.02
LassoLarsIC	0.52	0.53	2.06	2.93
LassoCV	0.52	0.53	2.06	24.09
BayesianRidge	0.52	0.52	2.07	3.28
RidgeCV	0.52	0.52	2.07	3.93
Ridge	0.52	0.52	2.07	1.25
TransformedTargetRegressor	0.52	0.52	2.07	2.89
ElasticNetCV	0.52	0.52	2.07	26.04
OrthogonalMatchingPursuitCV	0.51	0.52	2.08	5.30
OrthogonalMatchingPursuit	0.51	0.52	2.08	2.71
HuberRegressor	0.51	0.52	2.09	31.37
SGDRegressor	0.51	0.51	2.10	7.13
KNeighborsRegressor	0.50	0.51	2.10	10.85
LinearSVR	0.50	0.51	2.11	89.91
MLPRegressor	0.50	0.51	2.11	706.13
PoissonRegressor	0.49	0.50	2.12	3.41
LarsCV	0.49	0.49	2.14	9.02
GammaRegressor	0.43	0.43	2.26	3.17
TweedieRegressor	0.42	0.43	2.26	2.95
ElasticNet	0.40	0.40	2.31	1.32

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
LassoLars	0.37	0.38	2.37	1.27
DecisionTreeRegressor	0.30	0.30	2.50	35.65
ExtraTreeRegressor	0.27	0.27	2.56	8.06
AdaBoostRegressor	0.25	0.26	2.58	183.24
DummyRegressor	-0.01	-0.00	3.00	0.90
PassiveAggressiveRegressor	-0.42	-0.40	3.55	2.34
Lars	-4.39	-4.35	6.93	1.70
RANSACRegressor	-48.24	-47.84	20.94	5.17

```
models[models['RMSE'] < 2.0]
```

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
ExtraTreesRegressor	0.70	0.70	1.63	4902.69
BaggingRegressor	0.65	0.65	1.77	218.93
XGBRegressor	0.63	0.63	1.82	5.80
HistGradientBoostingRegressor	0.60	0.60	1.90	9.92
LGBMRegressor	0.60	0.60	1.90	7.96

```
models[ (models['RMSE'] <= 2.1) & (models['Time Taken'] < 10) ]
```

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
XGBRegressor	0.63	0.63	1.82	5.80
HistGradientBoostingRegressor	0.60	0.60	1.90	9.92
LGBMRegressor	0.60	0.60	1.90	7.96

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
LassoLarsCV	0.52	0.53	2.06	7.02
LassoLarsIC	0.52	0.53	2.06	2.93
BayesianRidge	0.52	0.52	2.07	3.28
RidgeCV	0.52	0.52	2.07	3.93
Ridge	0.52	0.52	2.07	1.25
TransformedTargetRegressor	0.52	0.52	2.07	2.89
OrthogonalMatchingPursuitCV	0.51	0.52	2.08	5.30
OrthogonalMatchingPursuit	0.51	0.52	2.08	2.71
SGDRegressor	0.51	0.51	2.10	7.13

Based on the RMSE scores and the computational time required, the following models have been chosen for subsequent analysis:

- `XGBRegressor`
- `HistGradientBoostingRegressor`
- `LGBMRegressor`

The selected models are all gradient boosting algorithms recognized for their ability to handle large-scale datasets, capture both linear and nonlinear patterns, and deliver high accuracy in complex predictive tasks. These models will undergo hyperparameter tuning to optimize their performance.

Feature Importance Based on SHAP Values

Based on the previous LazyPredict results, we selected the following models as the most promising candidates for further analysis:

- XGBRegressor
- HistGradientBoostingRegressor
- LGBMRegressor

In the next steps, we will tune these models using cross-validation on the preprocessed training data. Custom-developed tuners, tailored for different types of regressors, will be employed to optimize hyperparameters and improve model performance. Additionally, SHAP (SHapley Additive exPlanations) values will be utilized to assess feature importance and understand the contribution of individual features to the predictions.

```
# Import requiered Libraries
import os
import pandas as pd
from tqdm import tqdm

# Import custom classes for hyper-parameter tunning
# NOTE: Below, PATH_TO_PROJECT is the location of the project folder (e.g: /home/se
import sys
PATH_TO_SRC = 'PATH_TO_PROJECT/sep24_bds_int_medical'
sys.path.append( PATH_TO_SRC )

from src.features.tuners.XGBHyperparameterTuner import XGBHyperparameterTuner
from src.features.tuners.LGBMHyperparameterTuner import LGBMHyperparameterTuner
from src.features.tuners.HistGradientBoostingHyperparameterTuner import HistGradien

# Import the preprocessign pipeline
from pipelines import *

# Disable warnings
import warnings
warnings.filterwarnings("ignore")

# Import metrics for scoring
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer, mean_squared_error, root_mean_squared_erro
rmse_scorer = make_scorer(mean_squared_error, greater_is_better=False, squared=Fals

# Libraries for Plotting
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# Read and display the train.csv and test.csv data
data_dir = '../../../../data/'
train_file = os.path.join( data_dir , 'raw' , 'train.csv' )
test_file = os.path.join( data_dir , 'raw' , 'train.csv' )

df_train = pd.read_csv(train_file, index_col = 0 , parse_dates = True )
df_test = pd.read_csv(test_file, index_col = 0 , parse_dates = True )
```

Data Preprocessing

Main steps

- Re-encoding the `timestamp` into a `day-phase`
- Dropping the following columns: `activity`, `carbs`, `steps` , `p_num` and `time`
- Imputing NANs in the remaining columns with interpolation and medians
- Two negative values in the `insulin` column replaced with `0`
- The column `day-phase` is re-encoded using `pd.get_dummies()`
- Finally, all columns were transformed using `StandardScaler`.

```
# Split the data into Features and Target variables,
# and Standarize the features with the preprocessing pipelines
X = df_train.drop( 'bg+1:00' , axis = 1 )
y = df_train['bg+1:00']

data_pipe = pipeline
Xs = data_pipe.fit_transform( X )
Xs_test = data_pipe.transform( df_test )

# fix column names - Needed for LGBM
import re
Xs.columns = [re.sub(r'[^a-zA-Z0-9_]', '_', col) for col in Xs.columns]
Xs_test.columns = [re.sub(r'[^a-zA-Z0-9_]', '_', col) for col in Xs_test.columns]

# Display descriptive statistics
display( Xs.describe() )
display( Xs_test.describe() )
```

	bg_5_55	bg_5_50	bg_5_45	bg_5_40	bg_5_35
count	1.770240e+05	1.770240e+05	1.770240e+05	1.770240e+05	1.770240e+05
mean	6.678999e-17	7.128547e-16	1.746815e-16	-1.066071e-16	-8.862518e-17
std	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00
min	-2.022989e+00	-2.023072e+00	-2.024274e+00	-2.025959e+00	-2.025008e+00
25%	-7.257353e-01	-7.256400e-01	-7.259481e-01	-7.264489e-01	-7.260141e-01
50%	-1.935287e-01	-1.933603e-01	-1.933016e-01	-1.933167e-01	-1.930933e-01
75%	5.382552e-01	5.385243e-01	5.390874e-01	5.397402e-01	5.396728e-01
max	6.492316e+00	6.493403e+00	6.498071e+00	6.504157e+00	6.501724e+00

8 rows × 293 columns

	bg_5_55	bg_5_50	bg_5_45	bg_5_40	bg_5_35	bg_
count	3644.000000	3644.000000	3644.000000	3644.000000	3644.000000	3644.0(
mean	0.184348	0.186197	0.186528	0.184169	0.180827	0.1
std	1.115628	1.113388	1.111815	1.107850	1.104784	1.1(
min	-1.823411	-2.023072	-1.891112	-2.025959	-1.958393	-1.9
25%	-0.625947	-0.592570	-0.592786	-0.593166	-0.603886	-0.5
50%	-0.060477	-0.060290	-0.060140	-0.060034	-0.093171	-0.0(
75%	0.804358	0.771397	0.774894	0.750772	0.750621	0.7
max	6.425790	6.426868	6.431490	6.293126	6.312981	6.3

8 rows × 293 columns

Model Tuning

The code below initializes three hyperparameter tuners for XGBoost, LightGBM, and Histogram-based Gradient Boosting, fits them to the dataset to optimize their parameters. It reuses then saved tuners as pickle files, extracts the best models for each algorithm, and evaluates their performance

using root mean squared error (RMSE) on the training dataset. Finally, we organize the RMSE results in a DataFrame and sort them to rank the models.

```
# Instantiate and fit the Tunners
# xgb_tuner = XGBHyperparameterTuner( search_space = 'default' )
# lgbm_tuner = LGBMHyperparameterTuner( search_space = 'default' )
# hist_tuner = HistGradientBoostingHyperparameterTuner( search_space = 'default' )

# xgb_tuner.fit(X=Xs, y=y)
# lgbm_tuner.fit(X=Xs, y=y)
# hist_tuner.fit(X=Xs, y=y)
```

```
# Fit the Tuners
# xgb_tuner.fit(X=Xs, y=y)
# lgbm_tuner.fit(X=Xs, y=y)
# hist_tuner.fit(X=Xs, y=y)
# lasso_tuner.fit(X=Xs, y=y)
# ridge_tuner.fit(X=Xs, y=y)
# knn_tuner.fit(X=Xs, y=y)
# dummy_reg.fit(X=Xs, y=y)
```

```python
# Save or Read results from hyper-parameter tunning
def save_pickle( obj , file_name ):

    import pickle
    # Save to pickle file
    file_path = f'./models/bayes_search_res/{file_name}.pkl'
    with open(file_path, 'wb') as f:
        pickle.dump(obj, f)

    f.close()

def read_pickle( file_name ):
    import pickle

    # Specify the path to your pickle file
    file_path = f'./models/bayes_search_res/{file_name}.pkl'

    # Open the pickle file in binary read mode and load the object
    with open(file_path, 'rb') as f:
        obj = pickle.load( f )

    f.close()

    return obj

# Save results
# save_pickle( xgb_tuner , 'xgb_tuner_res' )
# save_pickle( lgbm_tuner , 'lgbm_tuner_res' )
# save_pickle( hist_tuner , 'hist_tuner_res' )

# Read Results
xgb_tuner = read_pickle( 'xgb_tuner_res' )
lgbm_tuner= read_pickle( 'lgbm_tuner_res' )
hist_tuner = read_pickle( 'hist_tuner_res' )
```

```
# Extract Best Models
xgb_best_model = xgb_tuner.get_best_model()
lgbm_best_model = lgbm_tuner.get_best_model()
hist_best_model = hist_tuner.get_best_model()

# Print the CV RMSE for the best models
xgb_rmse = root_mean_squared_error( y_true = y , y_pred = xgb_best_model.predict( X
lgmb_rmse = root_mean_squared_error( y_true = y , y_pred = lgbm_best_model.predict(
hist_rmse = root_mean_squared_error( y_true = y , y_pred = hist_best_model.predict(

rmse_df = pd.DataFrame({
    'model':['xgb','lgbm','hist'],
    'rmse':[xgb_rmse,lgmb_rmse,hist_rmse]
})

rmse_df.sort_values( 'rmse' , inplace = True )
rmse_df
```

	model	rmse
1	lgbm	1.757358
0	xgb	1.880405
2	hist	1.932709

Model Evaluation: Real vs. Predicted Correlation & Residuals Distribution

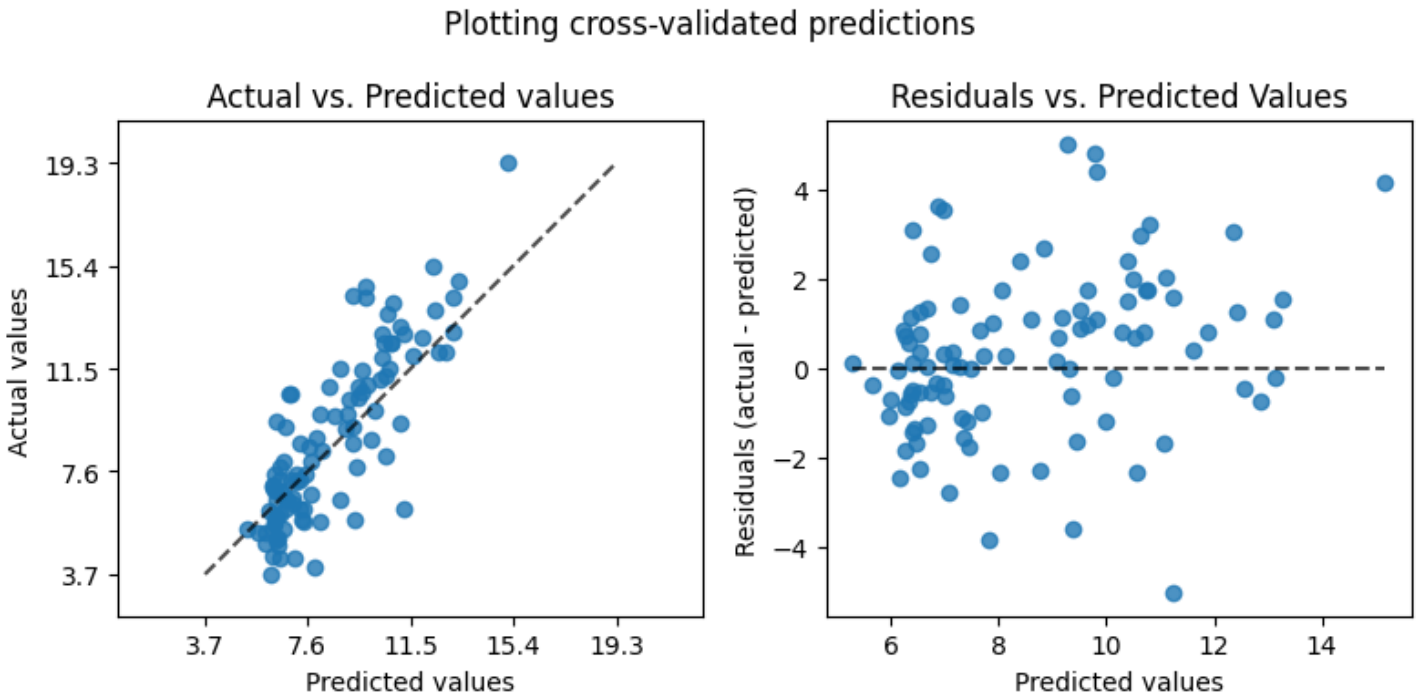
For each of the selected models, we evaluate their performance by analyzing the correlation between real and predicted values and the distribution of residuals. The plots below represent following insights:

- Real vs. Predicted Correlation: Shows how well the model's predictions align with the actual target values.
- Residuals Distribution: Indicates how errors are distributed, providing insight into bias and variance.

The following sections summarize these evaluations.

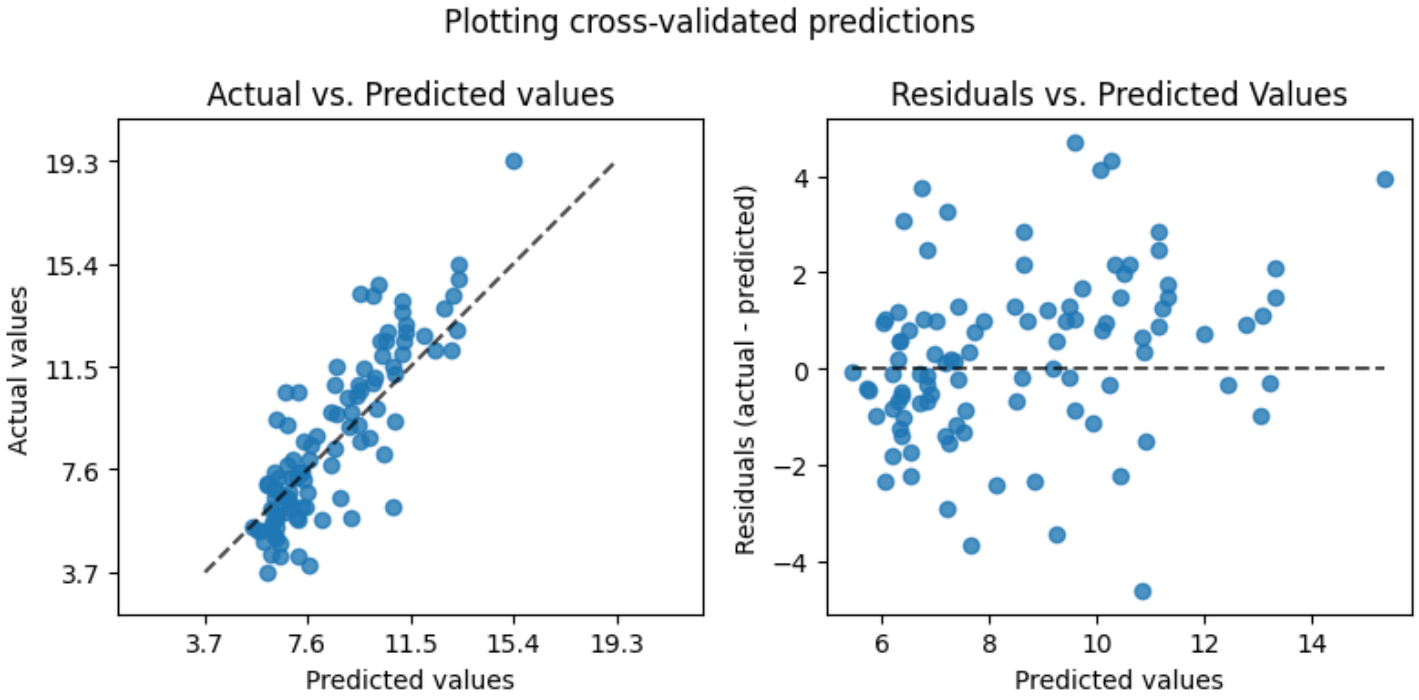
1. XGBoost Model

```
xgb_tuner.show_chart()
```



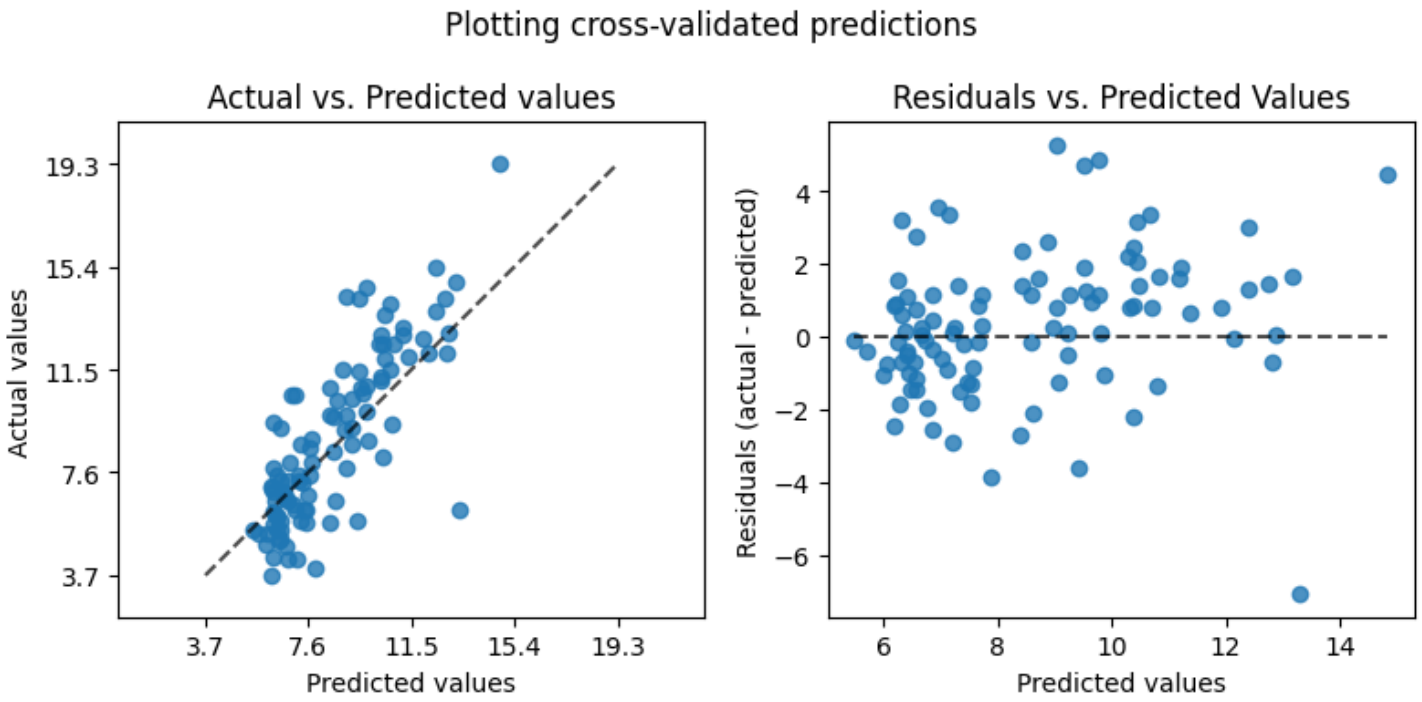
2. LGBM Model

```
lgbm_tuner.show_chart()
```



3. HistGradientBoosting Model

```
hist_tuner.show_chart()
```



Conclusion: All three models perform comparably well. The points in the correlation charts align reasonably well along the diagonal, indicating a strong correlation between actual and predicted values. However, some spread is observed, especially at higher values, indicating some underperformance in those areas. Looking at the residuals distributions, we can conclude that residuals are generally centered around zero, but show moderate dispersion, with occasional outliers. The LGBM model shows slightly more variability in residuals, while the HistGradientBoosting model shows greater dispersion and clustering, suggesting it may struggle to generalize in specific ranges.

Cross-Validated RMSE Scores for Top 100 Features

This section details the process of calculating cross-validated RMSE scores for Train/Test splits using the top 100 features identified through SHAP importance.

1. The function `train_test_cv` calculates RMSE scores for a model by iteratively evaluating its performance on subsets of the top 100 SHAP-ranked features.
2. Features are sorted by importance using the SHAP framework.
3. A 5-fold cross-validation is performed for each subset of features (from 1 to 100). Both train and test RMSE scores are calculated.

4. Train and test RMSE scores for each subset are stored in a results dataframe for comparison.
5. A dataframe containing RMSE scores for each feature subset, split by the model type.

```
def train_test_cv( tuner_res , name_tag):

    from src.features.helpers.ShapWrapper import ShapWrapper

    print(f'Calculating scores for {name_tag}')

    model = tuner_res.get_best_model()
    shap = ShapWrapper( model = model , X = Xs )
    sorted_list = shap.get_top_features()

    train_rmse_vals = []
    test_rmse_vals = []

    for n in tqdm( range(1,101) ):
        # Perform cross-validation, specifying both train and test scores
        cv_results = cross_validate(
            model, Xs[ sorted_list[:n] ], y, cv=5,
            scoring = rmse_scorer ,
            return_train_score=True
        )

        # Extract train and test scores
        train_scores = cv_results['train_score']
        test_scores = cv_results['test_score']

        # Display the results
        # Append final scores
        train_rmse_vals.append( -1*train_scores.mean() )
        test_rmse_vals.append( -1*test_scores.mean() )

        #Create a results df
        res = pd.DataFrame({
            'model':name_tag,
            'train_rmse':train_rmse_vals,
            'test_rmse':test_rmse_vals
        })

    return res
```

```
# Load base regressorsors

# xgb_rmse_vals = train_test_cv( xgb_tuner , 'XGBRegressor' )
# lgbm_rmse_vals = train_test_cv( lgbm_tuner , 'LGBMRegressor' )
# hist_rmse_vals = train_test_cv( hist_tuner , 'HistGradientBoostingRegressor' )

# rmse_vals_df = pd.concat([
#     xgb_rmse_vals.reset_index() ,
#     lgbm_rmse_vals.reset_index() ,
#     hist_rmse_vals.reset_index()
# ])

# rmse_vals_df.to_csv( './models/results_tables/01_RMSE_at different_topN_features.

#Read the global results table
rmse_vals_df = pd.read_csv( './models/results_tables/01_RMSE_at different_topN_feat
df_melt = rmse_vals_df.melt( id_vars=['index','model'] , var_name='type' , value_na
df_melt.head()
```

	index	model	type	rmse
0	0	XGBRegressor	train_rmse	2.122145
1	1	XGBRegressor	train_rmse	2.145826
2	2	XGBRegressor	train_rmse	2.037226
3	3	XGBRegressor	train_rmse	2.030764
4	4	XGBRegressor	train_rmse	2.004605

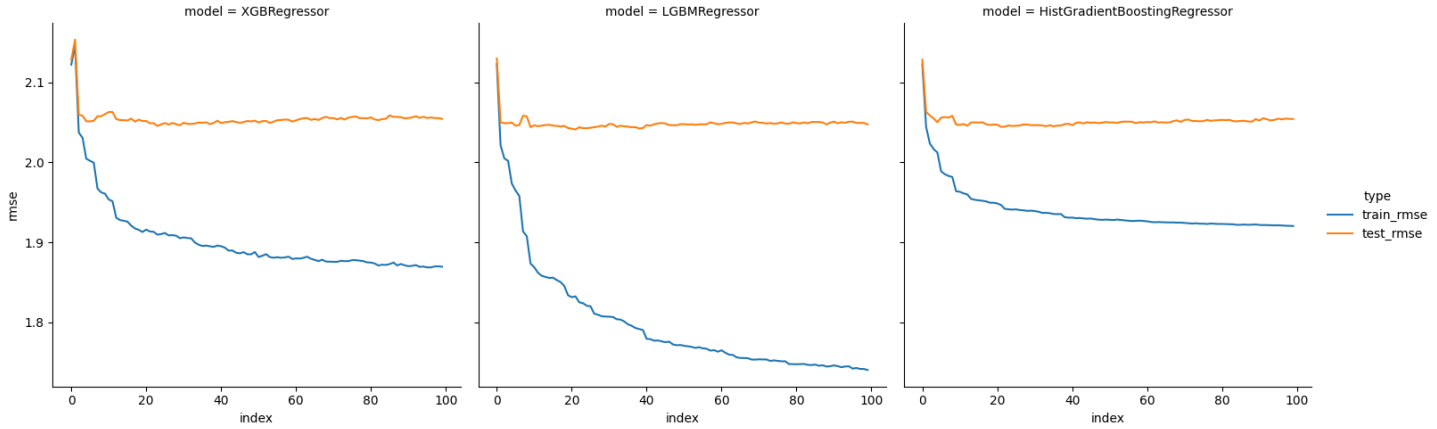
Plotting Model Performance Based on Top N Features

1. Performance for Top 100 Features

Using the RMSE scores for increasing numbers of SHAP-important features, the performance of each model is visualized to understand how adding more features affects model accuracy.

```
sns.relplot(
    data = df_melt ,
    x = 'index' ,
    y = 'rmse' ,
    hue = 'type' ,
    kind = 'line',
    col= 'model'
)
```

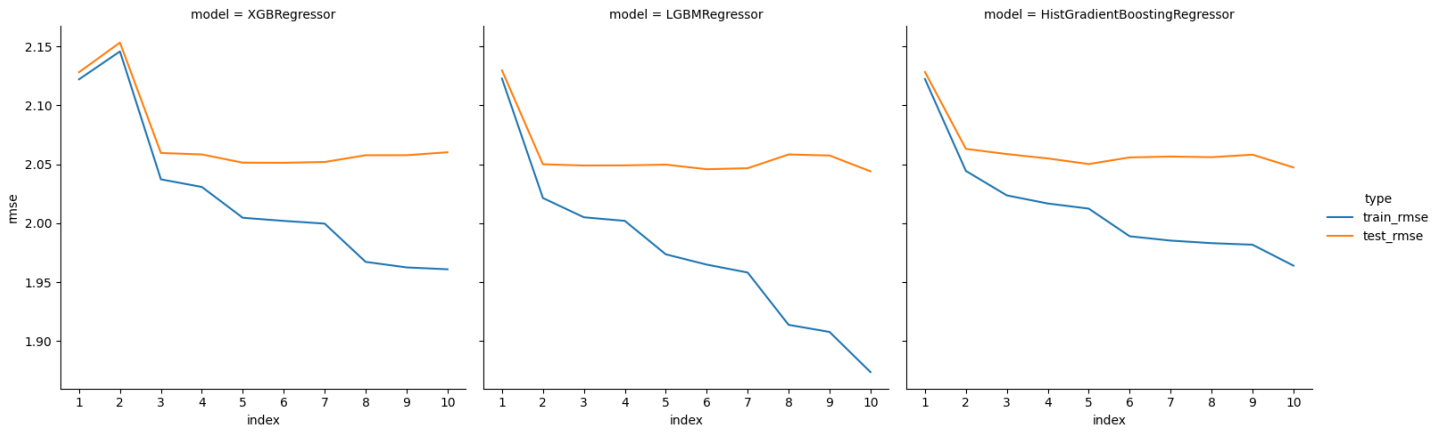
```
<seaborn.axisgrid.FacetGrid at 0x284398860>
```



2. Performance for Top 10 Features

A refined view of the performance is created by restricting the dataset to only the top 10 SHAP-important features.

```
g = sns.relplot(
    data = df_melt[ df_melt['index'] <= 9 ] ,
    x = 'index' ,
    y = 'rmse' ,
    hue = 'type' ,
    kind = 'line',
    col= 'model'
)

plt.xticks(
    ticks = list( range(10) ) ,
    labels = [x+1 for x in list( range(10) )]
    );
```



Observations:

- Top 100 features:
  - RMSE (blue) decreases significantly as more features are added, particularly within the first ~20-30 features.
  - Test RMSE (orange) also decreases with the addition of features but flatten after approximately 20-30 features, indicating that additional features provide little to no improvement in generalization performance.
  - All three models exhibit similar trends, with XGBRegressor and LGBMRegressor slightly outperforming HistGradientBoostingRegressor in terms of RMSE.
- Top 10 features:
  - Beyond 6-7 features, the train RMSE stabilizes, showing minimal benefit from additional features in this range.
  - Test RMSE shows a similar sharp decline with the first few features but stabilizes quickly, suggesting that only the top features have significant generalization power.
  - XGBRegressor appears to maintain slightly lower RMSE compared to LGBM and HistGradientBoosting, but the differences are minor.

Extracting the Top 10 Most Important Features

This code extracts the top 10 features for each model based on feature importance and identifies the common features across all three models.

```
def print_top_features( tuner , tag , n = 10 ):
    dfs = tuner.get_feature_importance()
    res_l = dfs['feature'][:n].tolist()
    print( f'Top_{n} { tag } fetures: \n{res_l}\n' )
    return res_l

xgboost_top_features = print_top_features( xgb_tuner , 'XGBoost' )
lgbm_top_features = print_top_features( lgbm_tuner , 'LGBM' )
hist_top_features =print_top_features( hist_tuner , 'HistGradientBoosting' )

common_top_feats = set(xgboost_top_features) & set(lgbm_top_features) & set(hist_to
common_top_feats = list(common_top_feats)
print( f'Common Top fetures: \n{common_top_feats}\n' )
```

```
Top_10 XGBoost fetures:
['bg_0_00', 'bg_0_05', 'bg_0_10', 'bg_0_15', 'hr_0_00', 'day_phase_night', 'bg_0_20

Top_10 LGBM fetures:
['bg_0_00', 'bg_0_05', 'bg_0_15', 'bg_0_10', 'hr_0_00', 'day_phase_night', 'day_phas

Top_10 HistGradientBoosting fetures:
['bg_0_00', 'bg_0_15', 'hr_0_00', 'day_phase_night', 'bg_0_10', 'insulin_0_00', 'ins

Common Top fetures:
['hr_0_00', 'bg_0_15', 'day_phase_evening', 'bg_0_00', 'insulin_0_00', 'day_phase_n
```

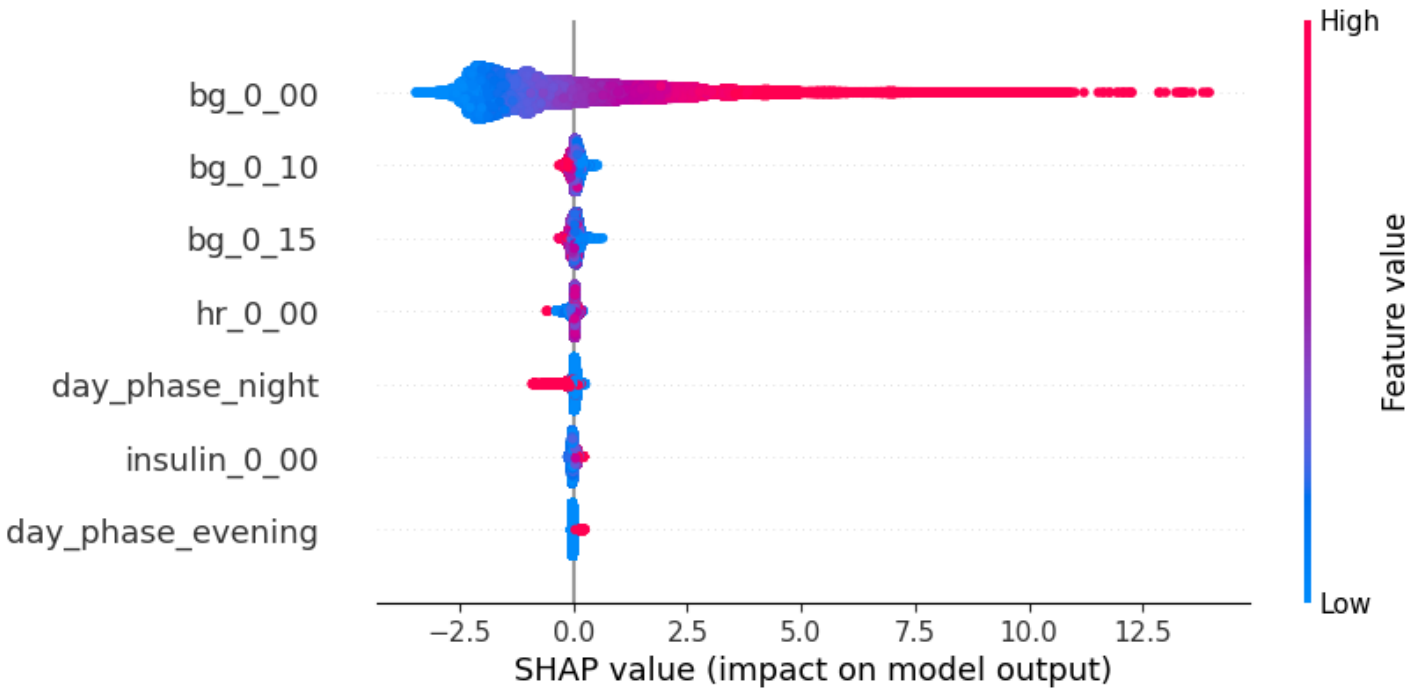
Beeswarm Plots of SHAP Values for Common Most-Important Features

The SHAP beeswarm plots display the impact of each feature on the model's predictions, highlighting how feature values contribute positively or negatively to the predicted outcomes. Using the common top features ensures a consistent comparison across models.

```
import shap

xgb_explainer = shap.Explainer( xgb_best_model  )
lgbm_explainer = shap.Explainer( lgbm_best_model )
hist_explainer = shap.Explainer( hist_best_model )

xgb_shap_values = xgb_explainer( Xs )
lgbm_shap_values = lgbm_explainer( Xs )
hist_shap_values = hist_explainer( Xs )
```

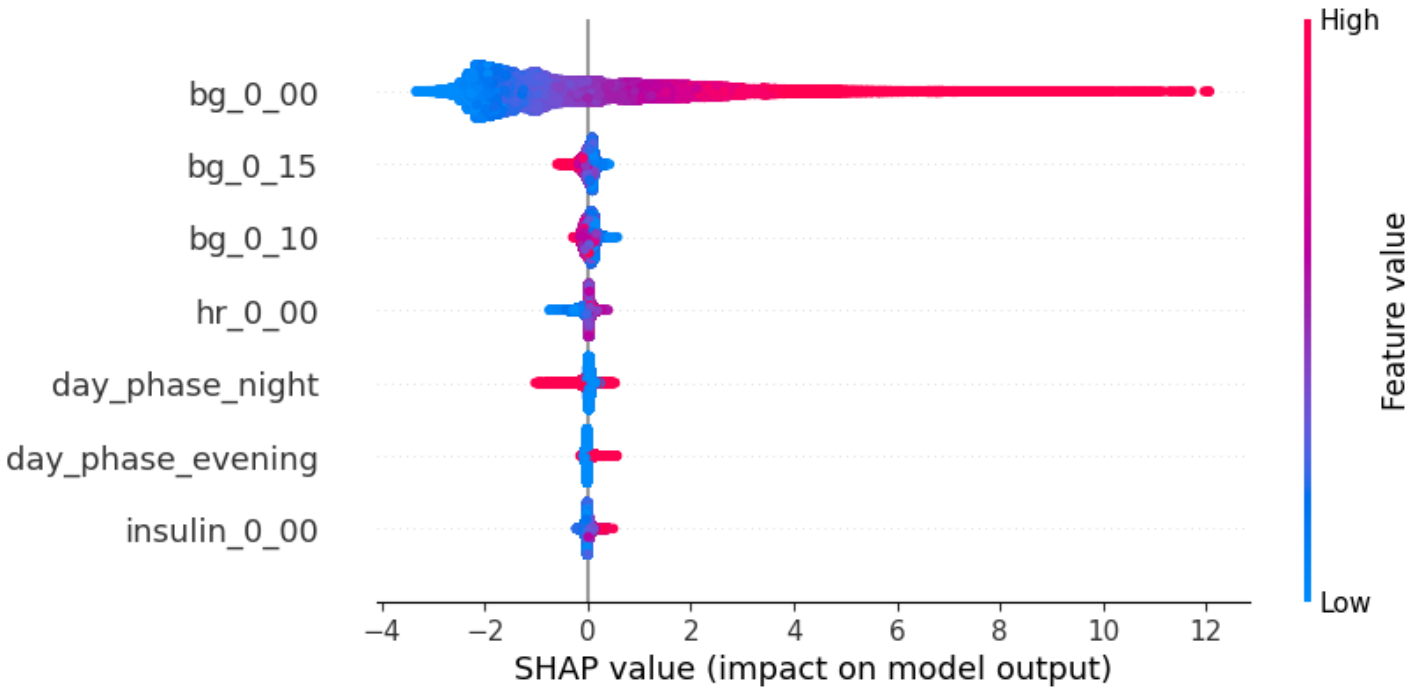
1. XGBoost Model - SHAP Values

```
shap.plots.beeswarm( xgb_shap_values[:,common_top_feats] , max_display = len( xgb_s
```



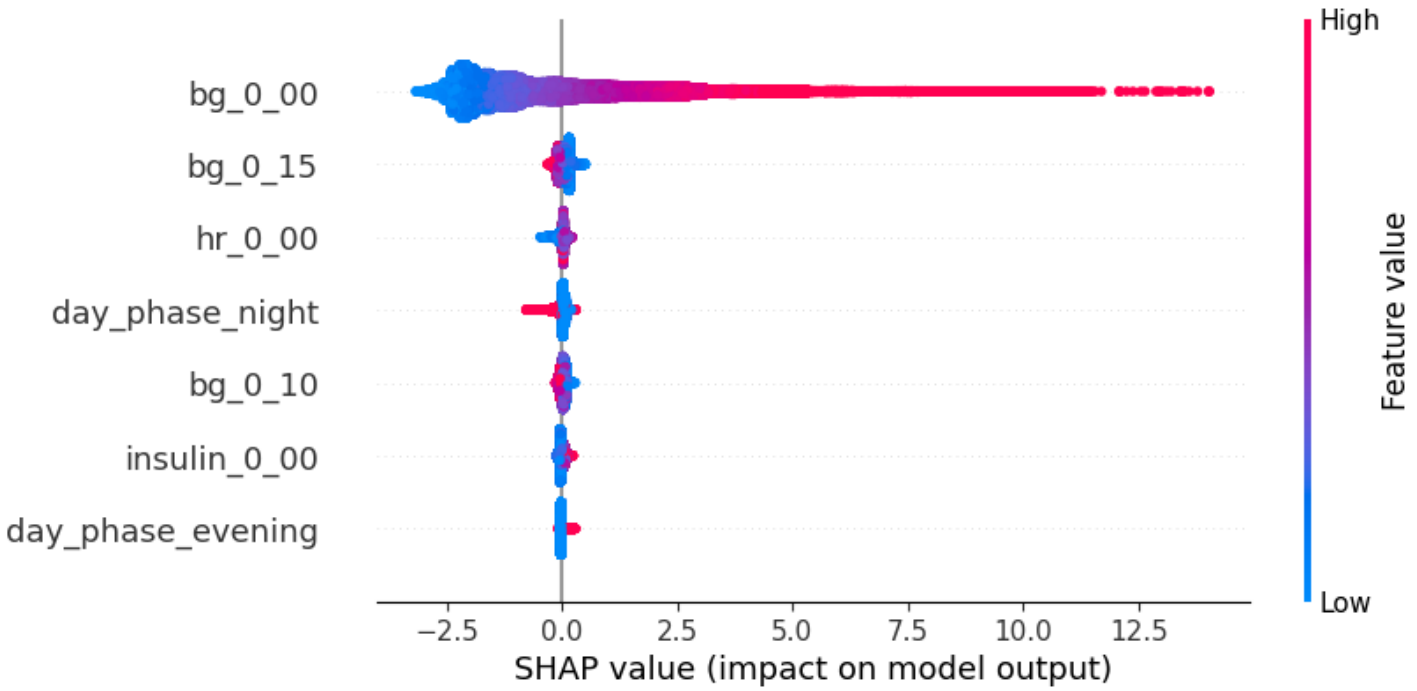
2. LGBM Model - SHAP Values

```
shap.plots.beeswarm( lgbm_shap_values[:,common_top_feats] )
```



3. HistGradientBoosting Model - SHAP Values

```
shap.plots.beeswarm( hist_shap_values[:,common_top_feats] )
```



Observations:

- Lagged glucose features `bg_0_00`, `bg_0_10`, `bg_0_15` are consistently the most impactful predictors.
- `day_phase` features (`day_phase_night`, `day_phase_evening`) and `hr_0_00` show less impactful, but still significant contributions across models.
- `insulin_0_00` also consistently contributes to predictions but with moderate SHAP impact.

Summary

- Based on RMSE scores, correlation plots, and residual analyses, all three tested models (**XGBoost**, **LGBM**, and **HistGradientBoosting**) perform reasonably well.
- The best performance after hyperparameter tunning can be achieved using the **Top 10 most important features** identified through SHAP values.
- For all the three tree-based models tested, **bg-0:00** (the most recent blood glucose reading) prominently emerges as the most important feature in predicting future glucose levels.

Re-running LazyPredict on the Feature Subset

Using the insights from the SHAP analysis in the previous chapter, we identified a subset of the most important features. In this step, we re-run LazyPredict on this refined feature subset to evaluate model performance and compare it with previous results.

```
# import get_lazy_regresor()
import importlib.util
import sys

def import_function_from_path(file_path, function_name):
    # Load the module from the file path
    spec = importlib.util.spec_from_file_location("module_name", file_path)
    module = importlib.util.module_from_spec(spec)
    sys.modules["module_name"] = module
    spec.loader.exec_module(module)

    # Retrieve the function from the loaded module
    func = getattr(module, function_name)
    return func

# NOTE: Below, PATH_TO_PROJECT is the location of the project folder (e.g: /home/se
PATH_TO_SCRIPT = 'PATH_TO_PROJECT/notebooks/helpers/LazyPredict.py'
function_name = 'get_lazy_regressor'

get_lazy_regressor = import_function_from_path( PATH_TO_SCRIPT , function_name )

PATH_TO_SRC = '/Users/masaver/Desktop/masaver/data_science_projects/sep24_bds_int_m
sys.path.append( PATH_TO_SRC )

# Load other requiered libraries
import os
import pandas as pd
from sklearn.model_selection import train_test_split

# Mute warnings
import warnings
warnings.filterwarnings("ignore")

# Import the preprocessign pipeline
from pipelines import *
```

```
# Read and display the train.csv and test.csv data
data_dir = '../../../../data/'
train_file = os.path.join( data_dir , 'raw' , 'train.csv' )
test_file = os.path.join( data_dir , 'raw' , 'train.csv' )

df_train = pd.read_csv(train_file, index_col = 0 , parse_dates = True )
df_test = pd.read_csv(test_file, index_col = 0 , parse_dates = True )
```

Data Preprocessing

Main steps

- Re-encoding the `timestamp` into a `day-phase`
- Dropping the following columns: `activity`, `carbs`, `steps` , `p_num` and `time`
- Imputing NANs in the remaining columns with interpolation and medians
- Two negative values in the `insulin` column replaced with `0`
- The column `day-phase` is re-encoded using `pd.get_dummies()`
- Finally, all columns were transformed using `StandardScaler`.

```
# Split the data into Features and Target variables,
# and Standarize the features with the preprocessing pipelines
X = df_train.drop( 'bg+1:00' , axis = 1 )
y = df_train['bg+1:00']

# fix column names
import re
X.columns = [re.sub(r'[^a-zA-Z0-9_]', '_', col) for col in X.columns]

# Train Test Split
x_train,x_test,y_train,y_test = train_test_split( Xs , y , test_size=0.2 , random_s
data_pipe = pipeline_s
x_train_s = data_pipe.fit_transform( x_train )
x_test_s = data_pipe.transform( x_test )

#Subset  to keep top features opnly
top_feat = ['hr_0_00', 'bg_0_15', 'day_phase_evening', 'bg_0_00', 'insulin_0_00', '
x_train_s = x_train_s[ top_feat ]
x_test_s = x_test_s[ top_feat ]

display( x_train_s )
display( x_test_s )
```

id	hr_0_00	bg_0_15	day_phase_evening	bg_0_00	insulin_0_00	day_pha
p11_1931	-0.18	0.74	-0.44	0.48	-0.12	
p10_9422	-0.34	-0.66	-0.44	-0.79	-0.12	
p06_8273	1.43	-1.20	-0.44	-1.29	-0.12	
p12_17133	1.87	0.34	-0.44	0.17	-0.12	
p03_2346	-0.61	0.04	-0.44	0.01	-0.12	
...	...	...	...	...	...	
p02_17172	-0.46	0.91	-0.44	0.84	-0.12	
p10_23964	0.47	1.18	-0.44	0.98	-0.12	
p03_7966	-1.00	-0.53	-0.44	-0.63	-0.12	
p03_628	-0.13	-0.49	-0.44	-0.53	-0.12	
p04_4394	2.97	-0.93	-0.44	-1.29	-0.12	

141619 rows × 7 columns

id	hr_0_00	bg_0_15	day_phase_evening	bg_0_00	insulin_0_00	day_pha
p11_17351	-0.13	0.01	-0.44	-0.09	-0.12	
p10_15385	-0.93	-0.86	-0.44	-0.79	-0.12	
p06_2496	-0.70	-0.06	-0.44	0.04	-0.12	
p10_8410	0.67	-0.03	-0.44	0.14	-0.12	
p02_2562	1.22	-0.49	-0.44	-0.36	-0.12	
...	...	...	...	...	...	
p02_24761	0.32	-0.63	-0.44	-0.69	-0.12	
p10_22597	0.11	0.54	2.26	0.58	0.53	
p03_2231	-1.55	-0.33	-0.44	-0.36	-0.12	
p12_4088	1.75	-0.36	2.26	-0.39	-0.11	
p04_19811	-0.82	1.94	-0.44	2.25	-0.12	

35405 rows × 7 columns

Run LazyPredict Regressor

```
# Run a Lazy Regressor
reg = get_lazy_regressor( exclude = ['SVR','QuantileRegressor'] )
models, predictions = reg.fit( x_train_s , x_test_s , y_train , y_test )
```

```
 97%|█████████▋| 36/37 [01:31<00:01,  1.13s/it]
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing wa
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1237
[LightGBM] [Info] Number of data points in the train set: 141619, number of used fea
[LightGBM] [Info] Start training from score 8.273489
```

```
100%|██████████| 37/37 [01:31<00:00,  2.47s/it]
```

```
display( models )
```

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
LGBMRegressor	0.55	0.55	2.00	0.43
HistGradientBoostingRegressor	0.55	0.55	2.01	0.92
MLPRegressor	0.55	0.55	2.01	53.60
XGBRegressor	0.55	0.55	2.01	0.40
GradientBoostingRegressor	0.54	0.54	2.04	6.23
SGDRegressor	0.52	0.52	2.08	0.17
Ridge	0.52	0.52	2.08	0.06
BayesianRidge	0.52	0.52	2.08	0.05
RidgeCV	0.52	0.52	2.08	0.05
TransformedTargetRegressor	0.52	0.52	2.08	0.05
LassoLarsCV	0.52	0.52	2.08	0.15
Lars	0.52	0.52	2.08	0.03
LarsCV	0.52	0.52	2.08	0.11
LassoLarsIC	0.52	0.52	2.08	0.09
LassoCV	0.52	0.52	2.08	0.43
ElasticNetCV	0.52	0.52	2.08	0.62
OrthogonalMatchingPursuitCV	0.51	0.51	2.09	0.10
HuberRegressor	0.51	0.51	2.09	0.33
LinearSVR	0.51	0.51	2.10	4.26
OrthogonalMatchingPursuit	0.49	0.49	2.15	0.09
PoissonRegressor	0.48	0.48	2.15	0.07
BaggingRegressor	0.48	0.48	2.16	2.86
KNeighborsRegressor	0.48	0.48	2.17	0.93
ExtraTreesRegressor	0.45	0.45	2.22	15.48
GammaRegressor	0.42	0.42	2.28	0.13
TweedieRegressor	0.42	0.42	2.28	0.07

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
ElasticNet	0.39	0.39	2.34	0.02
LassoLars	0.38	0.38	2.37	0.04
AdaBoostRegressor	0.29	0.29	2.52	2.49
ExtraTreeRegressor	0.11	0.11	2.82	0.27
DecisionTreeRegressor	0.11	0.11	2.83	0.51
DummyRegressor	-0.00	-0.00	3.00	0.01
PassiveAggressiveRegressor	-0.82	-0.82	4.05	0.16
RANSACRegressor	-10.42	-10.42	10.13	0.24

Conclusion: The previously selected models **XGBoost**, **LGBM** and **HistGradientBoosting** continue to be the best performers.

Hyperparameter Tuning on Feature Selection, Ensemble Regressor & Kaggle Submissions

In the following steps, we will focus on tuning the top-performing models (**XGBRegressor**, **LGBMRegressor**, and **HistGradientBoostingRegressor**) using the refined feature subset derived from the SHAP analysis.

```
# Import required Libraries
import os
import pandas as pd
from tqdm import tqdm

# Import custom classes for hyper-parameter tunning
# NOTE: Below, PATH_TO_PROJECT is the location of the project folder (e.g: /home/se
import sys

PATH_TO_SRC = 'PATH_TO_PROJECT/sep24_bds_int_medical'
sys.path.append(PATH_TO_SRC)

from src.features.tuners.XGBHyperparameterTuner import XGBHyperparameterTuner
from src.features.tuners.LGBMHyperparameterTuner import LGBMHyperparameterTuner
from src.features.tuners.HistGradientBoostingHyperparameterTuner import HistGradien
from src.features.tuners.LassoCVHyperparameterTuner import LassoCVHyperparameterTun
from src.features.tuners.RidgeHyperparameterTuner import RidgeHyperparameterTuner
from src.features.tuners.KNeighborsHyperparameterTuner import KNeighborsHyperparame

# Import Dummy Regressor
from sklearn.dummy import DummyRegressor

# Import the preprocessing pipeline
from pipelines import *

# Disable warnings
import warnings

warnings.filterwarnings("ignore")

# Import metrics for scoring
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer, mean_squared_error, root_mean_squared_erro

rmse_scorer = make_scorer(mean_squared_error, greater_is_better=False, squared=Fals

# Libraries for Plotting
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# Read train and test data
data_file = os.path.join('..', '..', '..', 'data', 'raw', 'train.csv')
test_file = os.path.join('..', '..', '..', 'data', 'raw', 'test.csv')

df_train = pd.read_csv(data_file, index_col=0, parse_dates=True)
df_test = pd.read_csv(test_file, index_col=0, parse_dates=True)
```

Data Preprocessing

Main steps

- Re-encoding the `timestamp` into a `day-phase`
- Dropping the following columns: `activity`, `carbs`, `steps` , `p_num` and `time`
- Imputing NANs in the remaining columns with interpolation and medians
- Two negative values in the `insulin` column replaced with `0`
- The column `day-phase` is re-encoded using `pd.get_dummies()`
- Finally, all columns were transformed using `StandardScaler`.

```
# Split the data into Features and Target variables,
# and Standarize the features with the preprocessing pipelines
X = df_train.drop('bg+1:00', axis=1)
y = df_train['bg+1:00']

data_pipe = pipeline
Xs = data_pipe.fit_transform(X)
Xs_test = data_pipe.transform(df_test)

# fix column names - Needed for LGBM
import re

Xs.columns = [re.sub(r'[^a-zA-Z0-9_]', '_', col) for col in Xs.columns]
Xs_test.columns = [re.sub(r'[^a-zA-Z0-9_]', '_', col) for col in Xs_test.columns]

#Subset  to keep top features opnly
top_feat = ['hr_0_00', 'bg_0_15', 'day_phase_evening', 'bg_0_00', 'insulin_0_00', '
Xs = Xs[top_feat]
Xs_test = Xs_test[top_feat]

# Display descriptive statistics
display(Xs.describe())
display(Xs_test.describe())
```

	hr_0_00	bg_0_15	day_phase_evening	bg_0_00	insulin_0_
count	1.770240e+05	1.770240e+05	1.770240e+05	1.770240e+05	1.770240e·
mean	7.809291e-16	-1.631217e-16	-3.901434e-17	-2.568846e-17	3.724826e
std	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e+00	1.000003e·
min	-2.447548e+00	-2.028078e+00	-4.444915e-01	-2.027635e+00	-1.94436
25%	-6.797723e-01	-7.267160e-01	-4.444915e-01	-7.263385e-01	-1.650581e
50%	-1.326035e-01	-1.928241e-01	-4.444915e-01	-1.924732e-01	-1.210919e
75%	4.807065e-01	5.412772e-01	-4.444915e-01	5.415915e-01	-8.54762
max	6.349242e+00	6.514193e+00	2.249762e+00	6.514209e+00	4.639869e

	hr_0_00	bg_0_15	day_phase_evening	bg_0_00	insulin_0_00	c
count	3644.000000	3644.000000	3644.000000	3644.000000	3644.000000	
mean	-0.060839	0.128647	0.020570	0.129674	0.005355	
std	0.890221	1.085989	1.018330	1.083849	0.960236	
min	-2.327291	-2.028078	-0.444492	-2.027635	-0.194436	
25%	-0.565528	-0.626611	-0.444492	-0.659605	-0.152482	
50%	-0.132604	-0.092719	-0.444492	-0.092374	-0.123205	
75%	0.259734	0.674750	-0.444492	0.675058	-0.089601	
max	5.140660	5.679987	2.249762	5.446479	24.957863	

Model Tuning

This process follows a similar methodology to the hyperparameter tuning previously conducted on the full feature set but tailored to the smaller, optimized subset of features. For this, the custom Tuners are used.

```
# Instantiate and fit the Tunners
# dummy_reg = DummyRegressor()
# xgb_tuner = XGBHyperparameterTuner( search_space = 'default' )
# lgbm_tuner = LGBMHyperparameterTuner( search_space = 'default' )
# hist_tuner = HistGradientBoostingHyperparameterTuner( search_space = 'default' )
# lasso_tuner = LassoCVHyperparameterTuner( search_space = 'default' )
# ridge_tuner = RidgeHyperparameterTuner( search_space = 'default' )
# knn_tuner  = KNeighborsHyperparameterTuner( search_space = 'default' )

# print('Fitting dummy regressor ... ')
# dummy_reg.fit(X=Xs, y=y)

# print('Fitting XGB Tuner ... ')
# xgb_tuner.fit(X=Xs, y=y)

# print('Fitting LGBM regressor ... ')
# lgbm_tuner.fit(X=Xs, y=y)

# print('Fitting HistGradientBoosting regressor ... ')
# hist_tuner.fit(X=Xs, y=y)

# print('Fitting LassoCV regressor ... ')
# lasso_tuner.fit(X=Xs, y=y)

# print('Fitting RidgeCV regressor ... ')
# ridge_tuner.fit(X=Xs, y=y)

# print('Fitting KNN regressor ... ')
# knn_tuner.fit(X=Xs, y=y)
```

```python
# Save or Read results from hyper-parameter tunning
def save_pickle(obj, file_name):
    import pickle
    # Save to pickle file
    file_path = f'./models/bayes_search_res/{file_name}.pkl'
    with open(file_path, 'wb') as f:
        pickle.dump(obj, f)

    f.close()


def read_pickle(file_name):
    import pickle

    # Specify the path to your pickle file
    file_path = f'./models/bayes_search_res/{file_name}.pkl'

    # Open the pickle file in binary read mode and load the object
    with open(file_path, 'rb') as f:
        obj = pickle.load(f)

    f.close()

    return obj


# # Save results
# save_pickle( xgb_tuner , 'xgb_tuner_res' )
# save_pickle( lgbm_tuner , 'lgbm_tuner_res' )
# save_pickle( hist_tuner , 'hist_tuner_res' )
# save_pickle( lasso_tuner , 'lasso_tuner_res' )
# save_pickle( ridge_tuner , 'ridge_tuner_res' )
# save_pickle( knn_tuner , 'knn_tuner_res' )
# save_pickle( dummy_reg , 'dummy_reg_res' )

# # Read Results
xgb_tuner = read_pickle('xgb_tuner_res')
lgbm_tuner = read_pickle('lgbm_tuner_res')
hist_tuner = read_pickle('hist_tuner_res')
lasso_tuner = read_pickle('lasso_tuner_res')
ridge_tuner = read_pickle('ridge_tuner_res')
knn_tuner = read_pickle('knn_tuner_res')
dummy_reg = read_pickle('dummy_reg_res')
```

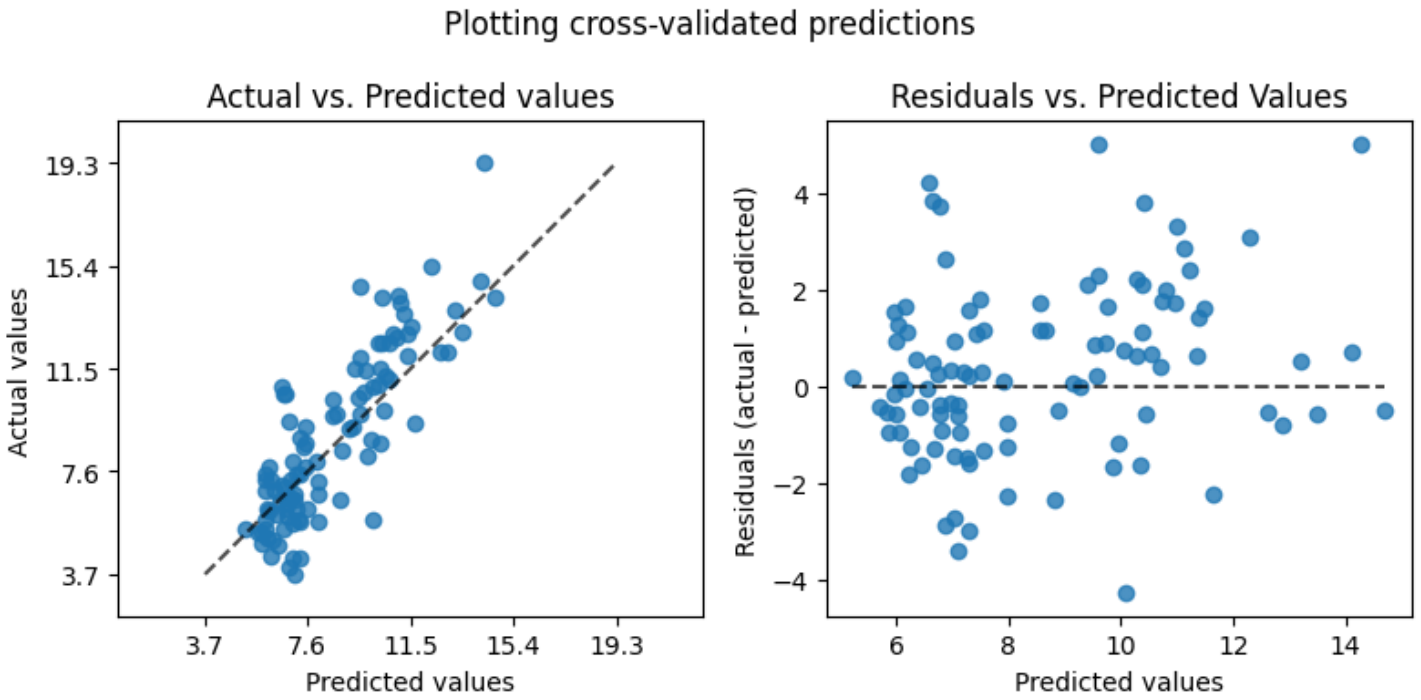
```
# Extract Best Models
xgb_best_model = xgb_tuner.get_best_model()
lgbm_best_model = lgbm_tuner.get_best_model()
hist_best_model = hist_tuner.get_best_model()
lasso_best_model = lasso_tuner.get_best_model()
ridge_best_model = ridge_tuner.get_best_model()
knn_best_model = knn_tuner.get_best_model()

# Print the CV RMSE for the best models
xgb_rmse = root_mean_squared_error(y_true=y, y_pred=xgb_best_model.predict(X=Xs))
lgmb_rmse = root_mean_squared_error(y_true=y, y_pred=lgbm_best_model.predict(X=Xs))
hist_rmse = root_mean_squared_error(y_true=y, y_pred=hist_best_model.predict(X=Xs))
lasso_rmse = root_mean_squared_error(y_true=y, y_pred=lasso_best_model.predict(X=Xs
ridge_rmse = root_mean_squared_error(y_true=y, y_pred=ridge_best_model.predict(X=Xs
knn_rmse = root_mean_squared_error(y_true=y, y_pred=knn_best_model.predict(X=Xs))
dummy_rmse = root_mean_squared_error(y_true=y, y_pred=dummy_reg.predict(X=Xs))

rmse_df = pd.DataFrame({
    'model': ['xgb', 'lgbm', 'hist', 'lasso', 'ridge', 'knn', 'dummy'],
    'rmse': [xgb_rmse, lgmb_rmse, hist_rmse, lasso_rmse, ridge_rmse, knn_rmse, dumm
})

rmse_df.sort_values('rmse', inplace=True)

display(rmse_df)
```

	model	rmse
5	knn	0.140675
1	lgbm	1.660832
0	xgb	1.827194
2	hist	1.832988
4	ridge	2.085131
3	lasso	2.085133
6	dummy	2.996390

Model Evaluation: Real vs. Predicted Correlation & Residuals Distribution

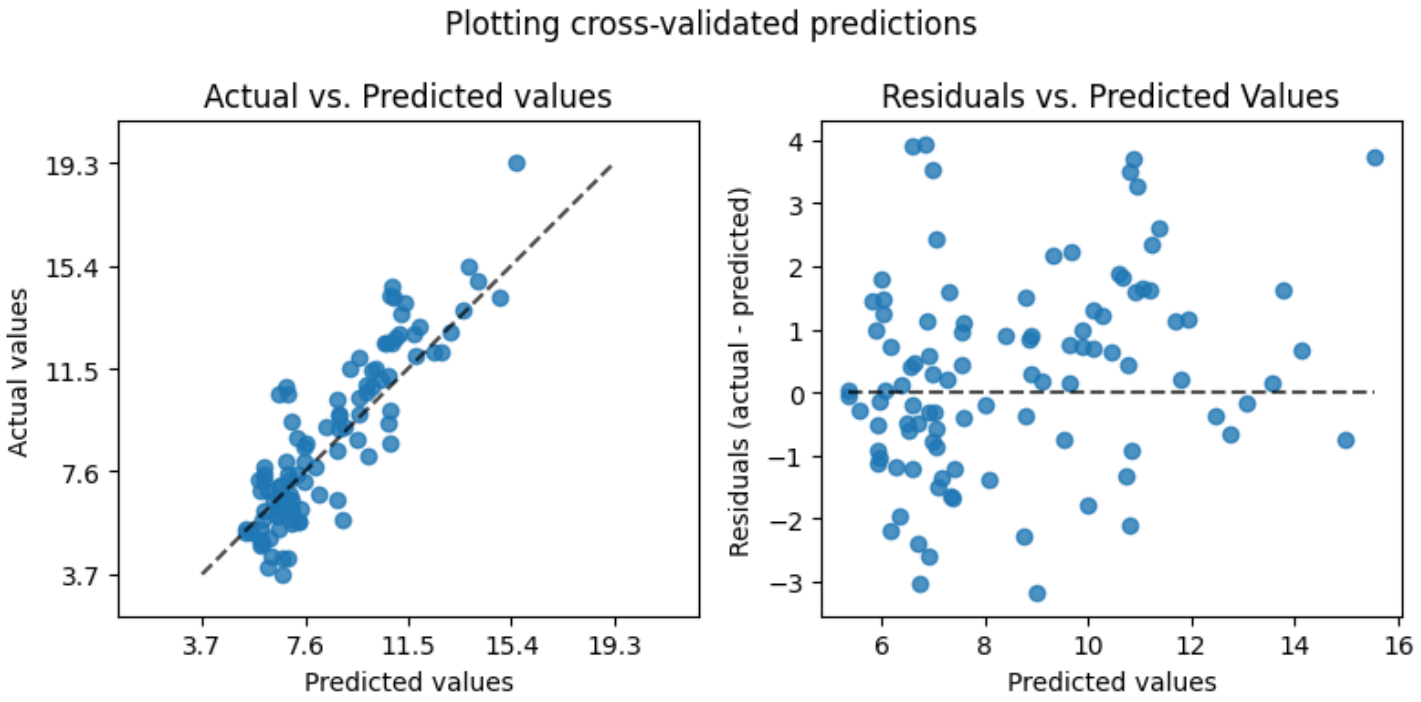
1. XGBoost Model

```
xgb_tuner.show_chart()
```



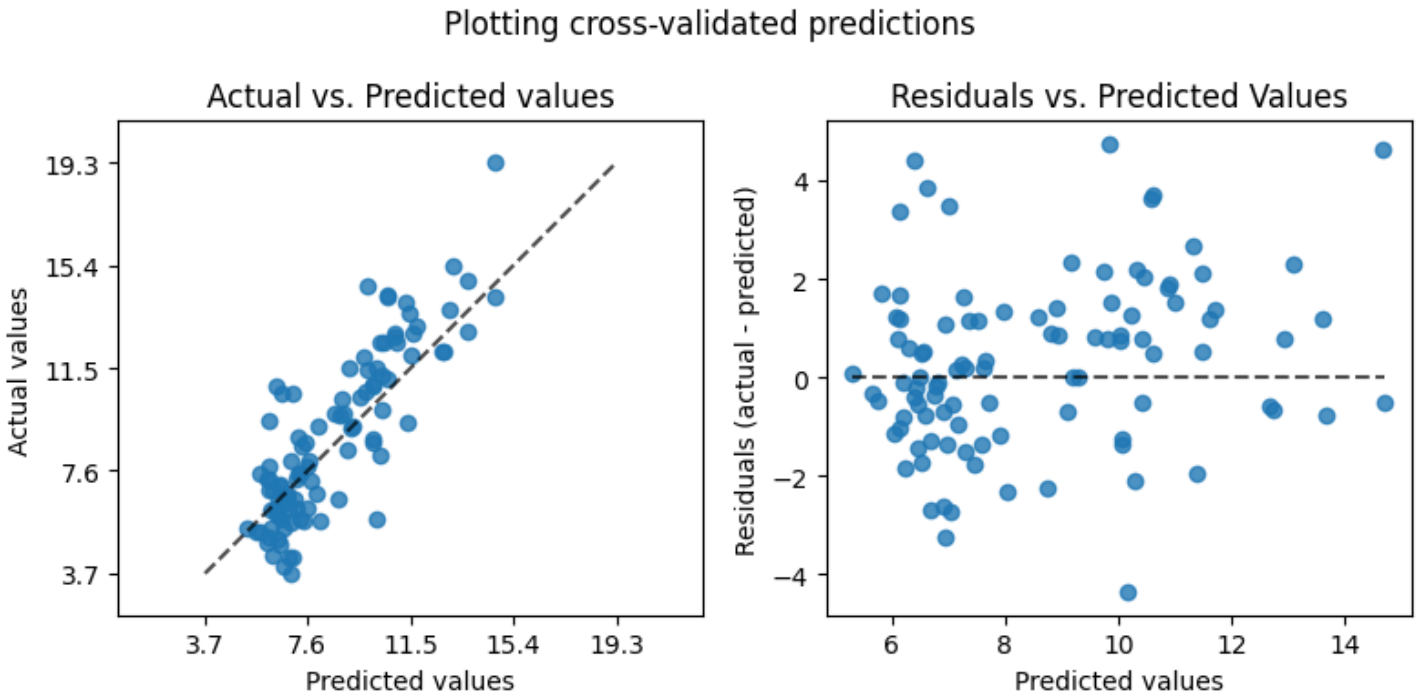
2. LGBM Model

```
lgbm_tuner.show_chart()
```



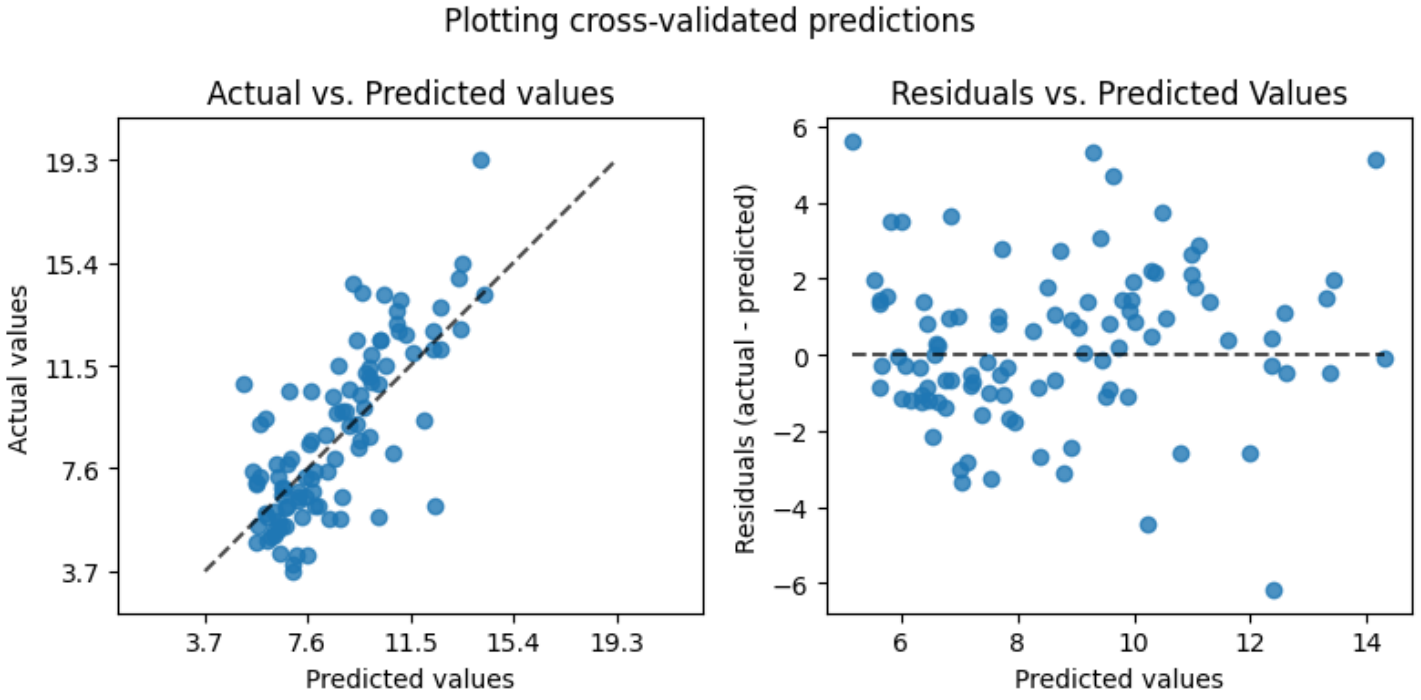
3. HistGradientBoosting Model

```
hist_tuner.show_chart()
```



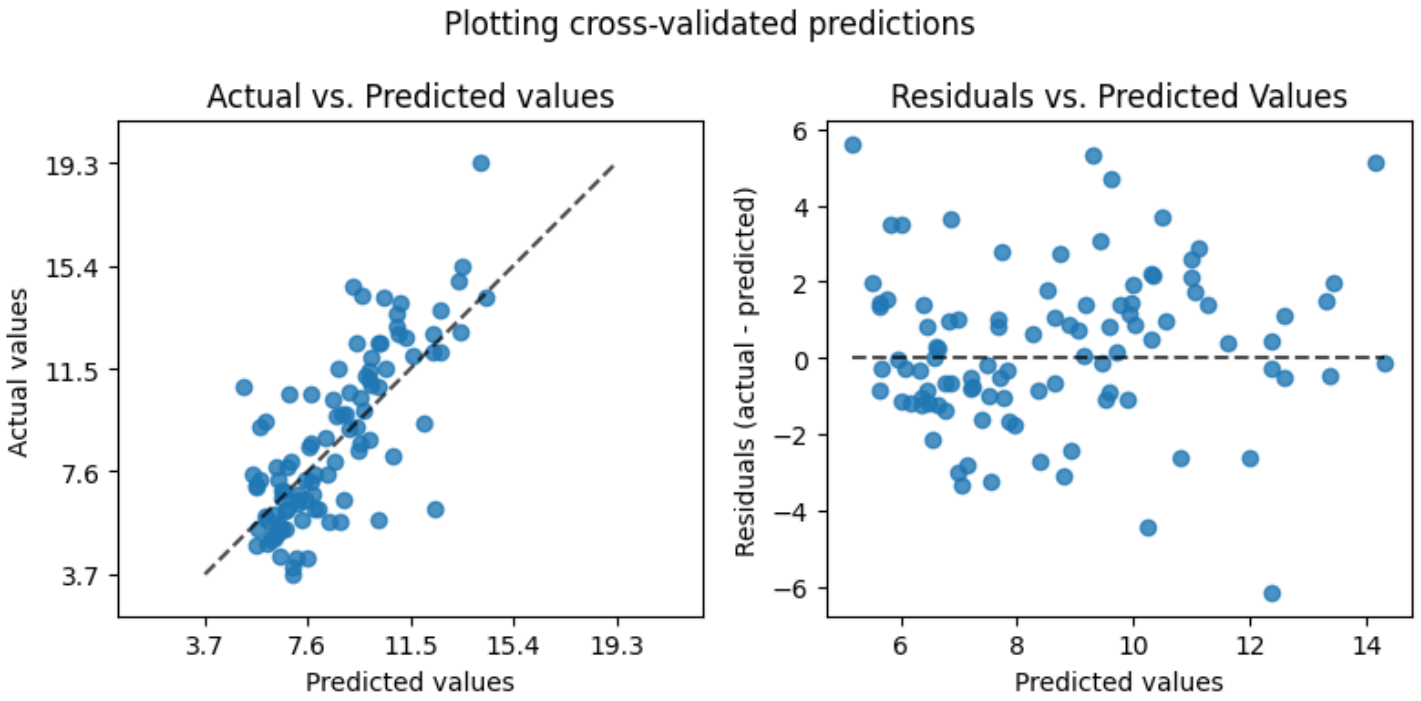
4. Lasso Model

```
lasso_tuner.show_chart()
```



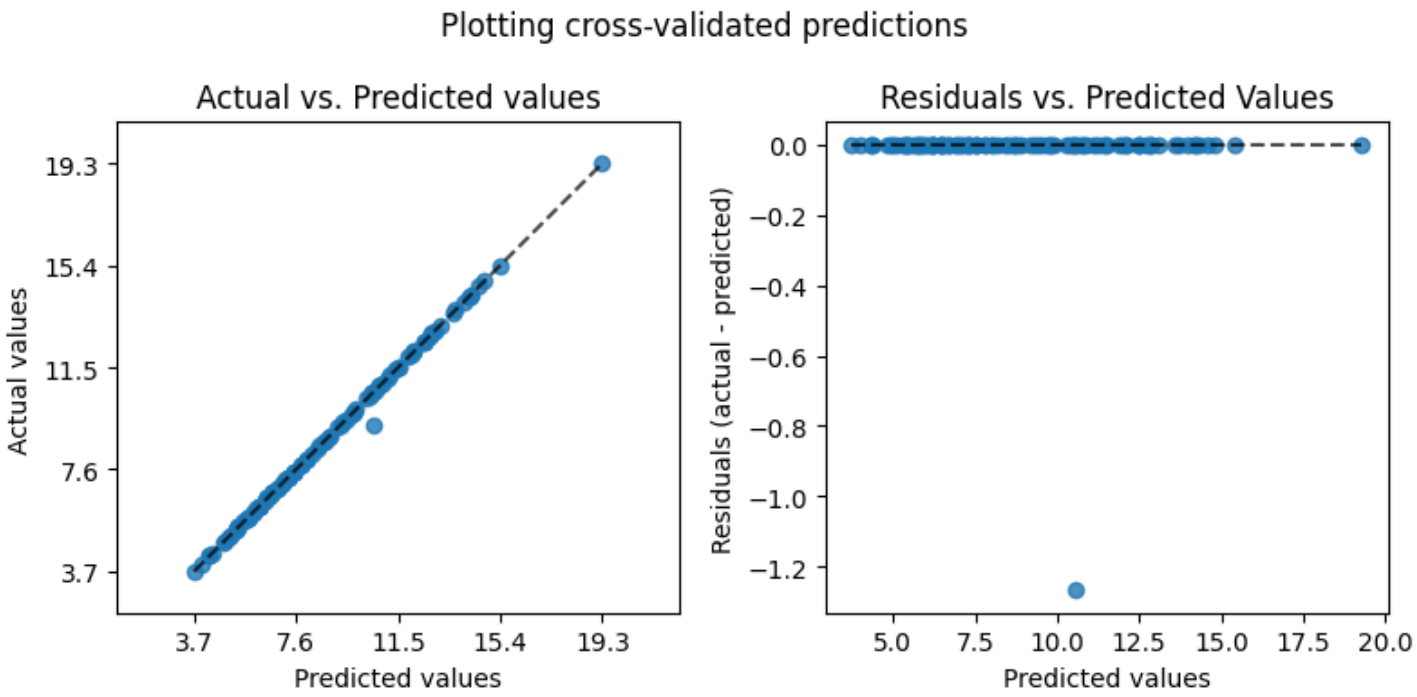
5. Ridge Model

```
ridge_tuner.show_chart()
```



6. KNN Model

```
knn_tuner.show_chart()
```



Cross-Validated RMSE Scores for Top 100 Features

```python
def train_test_cv(model, name_tag):
    from src.features.helpers.ShapWrapper import ShapWrapper

    print(f'Calculating scores for {name_tag}')

    # Perform cross-validation, specifying both train and test scores
    cv_results = cross_validate(
        model, Xs, y, cv=5,
        scoring=rmse_scorer,
        return_train_score=True
    )

    # Extract train and test scores
    train_scores = cv_results['train_score']
    test_scores = cv_results['test_score']

    #Create a results df
    res = pd.DataFrame([{
        'model': name_tag,
        'train_rmse': -1 * train_scores.mean(),
        'test_rmse': -1 * test_scores.mean()
    }])

    return res
```

```
xgb_rmse_vals = train_test_cv(knn_best_model, 'XGBoost')
lgbm_rmse_vals = train_test_cv(lgbm_best_model, 'LGBM')
hist_rmse_vals = train_test_cv(hist_best_model, 'HistGradient')
lasso_rmse_vals = train_test_cv(lasso_best_model, 'Lasso')
ridge_rmse_vals = train_test_cv(ridge_best_model, 'Ridge')
knn_rmse_vals = train_test_cv(knn_best_model, 'KNN')
dummy_rmse_vals = train_test_cv(dummy_reg, 'Dummy')

res = pd.concat([
    xgb_rmse_vals, lgbm_rmse_vals, hist_rmse_vals,
    lasso_rmse_vals, ridge_rmse_vals, knn_rmse_vals,
    dummy_rmse_vals
])

res.sort_values('test_rmse', inplace=True)
display(res)
```

```
Calculating scores for XGBoost
Calculating scores for LGBM
Calculating scores for HistGradient
Calculating scores for Lasso
Calculating scores for Ridge
Calculating scores for KNN
Calculating scores for Dummy
```

	model	train_rmse	test_rmse
0	Lasso	2.083482	2.086481
0	Ridge	2.083473	2.086483
0	HistGradient	1.802948	2.094037
0	XGBoost	0.134945	2.110250
0	KNN	0.134945	2.110250
0	LGBM	1.612102	2.126951
0	Dummy	2.993084	3.012515

Ensemble Model with VotingRegressor

In this step, we will combine the already tuned Regressors into an ensemble model using VotingRegressor.

> *Note*: Due to it's hihgh degree of overfitting, the KNN model is excluded from the ensemble model.

```
from sklearn.ensemble import VotingRegressor

# Create a VotingRegressor
voting_regressor = VotingRegressor(estimators=[
    ('lasso', lasso_best_model),
    ('ridge', ridge_best_model),
    ('hist', hist_best_model),
    ('xgb', xgb_best_model),
    ('lgbm', lgbm_best_model)
])

# Fit the ensemble regressor
voting_regressor.fit(Xs, y)

# Calculate CVed RMSE scores for the VotingRegressor
voting_rmse_vals = train_test_cv(voting_regressor, 'Voting Regressor')
voting_rmse_vals
```

```
Calculating scores for Voting Regressor
```

	model	train_rmse	test_rmse
0	Voting Regressor	1.833329	2.056399

Conclusions:

- The ensemble model achieves a lower test RMSE (2.056) compared to all individual models.
- The train RMSE (1.833) is reasonable and indicates reduced overfitting compared to individual models like XGBoost and KNN.

Prepare Kaggle Submission

```python
def prep_submission_table(model, tag):
    print(f'Predicitng with the {tag} model ... \n')
    pred_df = pd.DataFrame({
        'id': Xs_test.index,
        'bg+1:00': model.predict(Xs_test)
    })

    output_name = f'./submissions/{tag}_res.csv'
    pred_df.to_csv(output_name, index=False)
    # display( pred_df )


prep_submission_table(dummy_reg, 'Dummy_Reg')
prep_submission_table(voting_regressor, 'Voting_Reg')
prep_submission_table(lasso_best_model, 'Lasso_Reg')
prep_submission_table(ridge_best_model, 'Ridge_Reg')
prep_submission_table(hist_best_model, 'Hist_Reg')
prep_submission_table(xgb_best_model, 'XGBoost_Reg')
prep_submission_table(knn_best_model, 'KNN_Reg')
prep_submission_table(lgbm_best_model, 'LGBM_Reg')
```

```
Predicitng with the Dummy_Reg model ...

Predicitng with the Voting_Reg model ...

Predicitng with the Lasso_Reg model ...

Predicitng with the Ridge_Reg model ...

Predicitng with the Hist_Reg model ...

Predicitng with the XGBoost_Reg model ...

Predicitng with the KNN_Reg model ...

Predicitng with the LGBM_Reg model ...
```

Kaggle Score Results

For each of the tested models, predictions were generated and submitted to the competition on Kaggle. Below are the **expected RMSE scores** (from cross-validation) and the **Kaggle leaderboard scores**:

Model	Expected RMSE	Kaggle Score
Lasso	2.08	2.62
Ridge	2.08	2.62
XGBoost	2.11	2.59
VotingRegressor	2.05	2.56

Summary

- Based on RMSE, correlation, and residual plots, the individual models (Lasso, Ridge, and XGBoost) performed reasonably well when trained on the top important features derived from SHAP analysis.
- Combining the individual models into a VotingRegressor ensemble produces slightly better results to the individual models.
- Kaggle scores were higher (worse) than the RMSE values obtained during our work, obviously due to differences in data distribution in the test set used by Kaggle.

Model 2 - Data Augmentation

Overview

This notebook explores a data augmentation approach, where the given test dataset is used to generate additional training data. By leveraging predicted blood glucose levels, we augment the dataset to enhance the model's training process and improve its performance when predicting the test data.

Objective:

To assess whether data augmentation via test data predictions can improve model performance by providing additional information for training.

Data Augmentation

Data augmentation is a technique used to increase the size of the training dataset based on existing data or newly created synthetic data from existing data. In this case, we will use the lag features given in the test data to create more input data for the model.

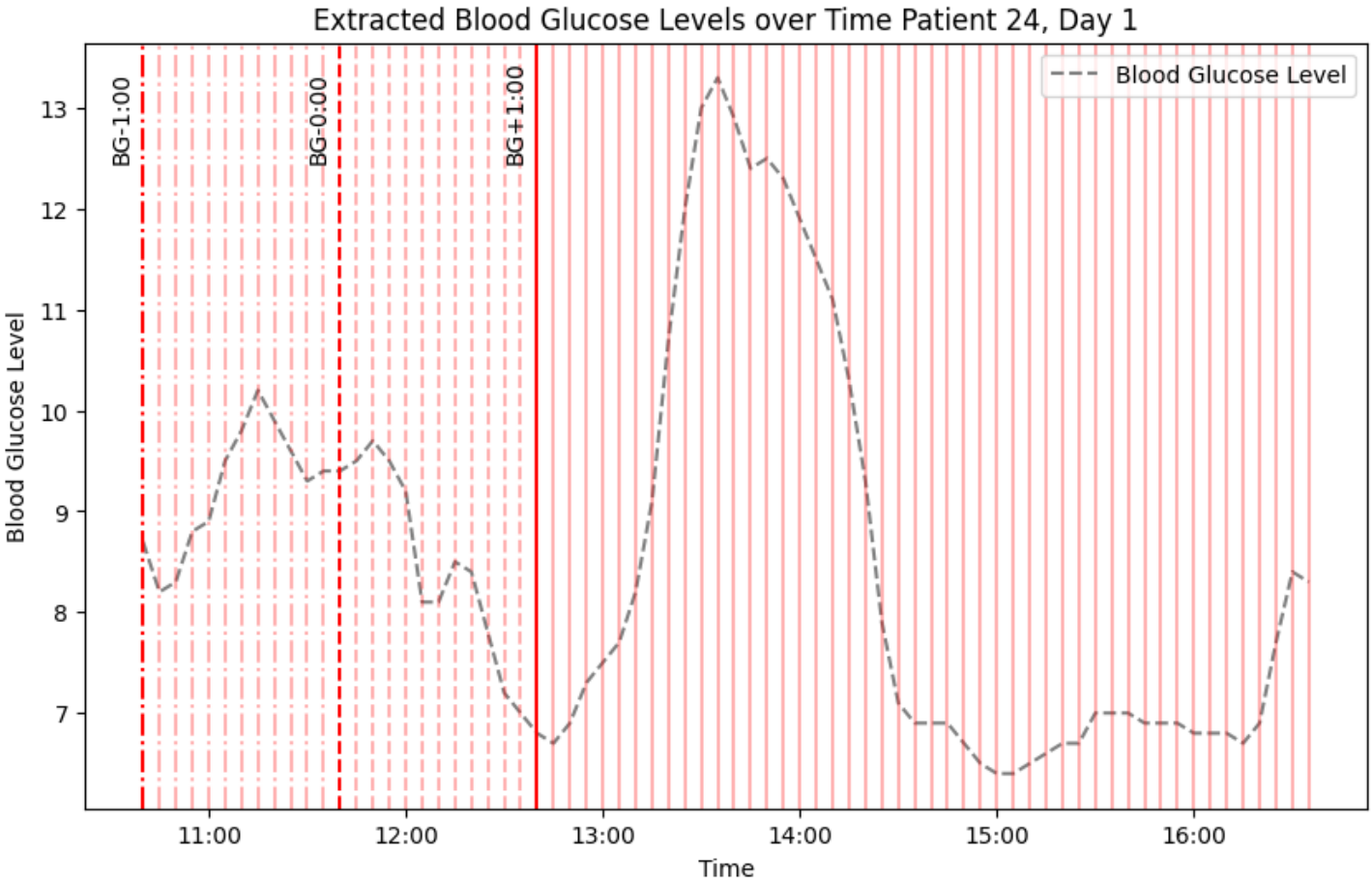
For each imputed row in the test set we can create a new row by shifting the lag features by one column. This will create a new row with the same target value but with different input features. We can then use this new data to train the model.

```
from datetime import datetime

from src.features.helpers.extractors import extract_patient_data
import os
import pandas as pd

df_test = pd.read_csv(os.path.join('..', '..', '..', 'data', 'raw', 'test.csv'))
extracted_data = extract_patient_data(df_test, 'p24')
extracted_data
```

	p_num	time	bg	insulin	carbs	hr	steps	cals	activity
2020-01-02 10:40:00	p24	10:40:00	8.7	0.0708	NaN	79.1	0.0	4.39	NaN
2020-01-02 10:45:00	p24	10:45:00	8.2	0.0708	NaN	84.1	225.0	15.94	NaN
2020-01-02 10:50:00	p24	10:50:00	8.3	0.0708	NaN	104.3	41.0	12.11	NaN
2020-01-02 10:55:00	p24	10:55:00	8.8	0.0708	NaN	85.0	53.0	6.97	NaN
2020-01-02 11:00:00	p24	11:00:00	8.9	0.0708	NaN	92.2	145.0	13.20	NaN
...	...	...	...	...	...	...	...	...	...
2020-11-29 02:50:00	p24	02:50:00	6.4	0.0771	NaN	72.7	0.0	4.31	NaN
2020-11-29 02:55:00	p24	02:55:00	6.4	0.0000	NaN	71.8	NaN	4.15	NaN
2020-11-29 03:00:00	p24	03:00:00	6.4	0.0000	NaN	76.2	0.0	4.23	NaN
2020-11-29 03:05:00	p24	03:05:00	6.5	0.0327	NaN	70.5	NaN	4.15	NaN
2020-11-29 03:10:00	p24	03:10:00	6.4	0.0771	NaN	73.9	NaN	4.15	NaN

95527 rows × 9 columns

```python
# plot the extracted data for the first day
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

# show only data from the first day
extracted_data = extracted_data[extracted_data.index < datetime(2020, 1, 3)]
plt.figure(figsize=(10, 6))

plt.plot(extracted_data.index, extracted_data['bg'], linestyle='--', color='grey',
plt.xlabel('Time')
plt.ylabel('Blood Glucose Level')
plt.title('Extracted Blood Glucose Levels over Time Patient 24, Day 1')

date_format = mdates.DateFormatter("%H:%M")
plt.gca().xaxis.set_major_formatter(date_format)

# Vertical line at 18:00 with label 'BG+1h'
plt.axvline(x=datetime(2020, 1, 2, 10, 40), color='r', linestyle='-.')
plt.text(datetime(2020, 1, 2, 10, 30), 12.5, 'BG-1:00', rotation=90)

start_date = datetime(2020, 1, 2, 10, 45)
for i in range(11):
    plt.axvline(x=start_date + pd.Timedelta(minutes=5 * i), color='r', linestyle='-

plt.axvline(x=datetime(2020, 1, 2, 11, 40), color='r', linestyle='--')
plt.text(datetime(2020, 1, 2, 11, 30), 12.5, 'BG-0:00', rotation=90)

start_date = datetime(2020, 1, 2, 11, 45)
for i in range(11):
    plt.axvline(x=start_date + pd.Timedelta(minutes=5 * i), color='r', linestyle='-

plt.axvline(x=datetime(2020, 1, 2, 12, 40), color='r', linestyle='-')
plt.text(datetime(2020, 1, 2, 12, 30), 12.5, 'BG+1:00', rotation=90)

start_date = datetime(2020, 1, 2, 12, 45)

for i in range(47):
    plt.axvline(x=start_date + pd.Timedelta(minutes=5 * i), color='r', linestyle='-

plt.legend()
plt.show()
```



With this method we can create more data from the existing test dataset. Depending on the number of lag features we use, more or less data can be created.

- For 1h lag: 47 new rows for each row in the test data (174.941 rows in total)
- For 2h lag: 35 new rows for each row in the test data (131,320 rows in total)
- For 3h lag: 23 new rows for each row in the test data (87.897 rows in total)
- For 4h lag: 11 new rows for each row in the test data (44711 rows in total)
- For 4:55h lag: 1 new rows for each row in the test data (3646 rows in total)

Preprocessing and Standardization

Pipelines

We created `preprocessing pipeline` based on the transformers already described in the previous sections. The following steps have been implemented:

- `DateTimeTransformer`: Extracts the time from the time column and creates circular features for the hour and minute
- `DropColumnsTransformer`: Drops all columns for parameters `activity` and `carbs`
- `FillPropertiesNaNsTransformer`: Interpolate (limit 3), forwards and backwards fill (limit 1) and median for the remaining columns for `bg`, `insulin`, `hr`, `steps`
- `DropOutliersTransformer`: Find an rewrite outliers for `insulin`
- `ExtractFeaturesTransformer`: Extracts all specified columns, here:
  - hour_sin, hour_cos
  - bg-0:00 - bg-2:00
  - insulin-0:00 - insulin-2:00
  - cals-0:00 - cals-2:00
  - hr-0:00 - hr-2:00
  - steps-0:00 - steps-2:00
  - p_num
  - bg+1:00 (target)

```
preprocessing_pipeline = Pipeline(steps=[
    ('date_time', DateTimeHourTransformer(time_column='time', result_column='hour',
    ('drop_parameter_cols', DropColumnsTransformer(starts_with=['activity', 'carbs']
    ('drop_others', DropColumnsTransformer(columns_to_delete=['time'])),
    ('fill_properties_nan_bg', FillPropertyNaNsTransformer(parameter='bg', how=['in
    ('fill_properties_nan_insulin', FillPropertyNaNsTransformer(parameter='insulin'
    ('fill_properties_nan_cals', FillPropertyNaNsTransformer(parameter='cals', how=
    ('fill_properties_nan_hr', FillPropertyNaNsTransformer(parameter='hr', how=['in
    ('fill_properties_nan_steps', FillPropertyNaNsTransformer(parameter='steps', ho
    ('drop_outliers', PropertyOutlierTransformer(parameter='insulin', filter_functi
    ('extract_features', ExtractColumnsTransformer(columns_to_extract=columns_to_ex
])
```

The `standardization pipeline` contains:

- `GetDummiesTransformer`: One-hot encodes the `p_num` column
- `StandardScaler`: Standardizes the data (excluding the target column)

```
standardization_pipeline = Pipeline(steps=[
  ('get_dummies', GetDummiesTransformer(columns=['hour', 'p_num'])),
    ('standard_scaler', StandardScalerTransformer(columns=columns_to_extract[3:-1])
])
```

```
import pandas as pd
from src.features.helpers.load_data import load_data
from src.models.model_2.model.pipelines_2h import pipeline

train_data, augmented_data, test_data = load_data('2_00h')

all_train_data_transformed = pipeline.fit_transform(pd.concat([train_data, augmente

X_train, y_train = all_train_data_transformed.iloc[len(train_data):].drop(columns=[
X_augmented, y_augmented = all_train_data_transformed.iloc[:len(train_data)].drop(c

all_train_data_transformed.head()
```

	hour_sin	hour_cos	bg-2:00	bg-1:55	bg-1:50	bg-1:45	bg-1:40	bg
id								
p01_0	0.999048	-0.043619	2.815442	2.915458	3.044933	3.141255	3.176766	3.1
p01_1	0.994056	-0.108867	3.138183	3.173915	3.174163	3.205851	3.176766	3.1
p01_2	0.984808	-0.173648	3.202732	3.173915	3.109548	3.076659	3.015126	2.9
p01_3	0.971342	-0.237686	3.073635	3.012379	2.915704	2.850574	2.885814	2.8
p01_4	0.953717	-0.300706	2.847716	2.883151	2.883396	2.915169	2.885814	2.8

5 rows × 142 columns

Preliminary Modelling and Model Selection

```
import pandas as pd
from src.features.helpers.load_data import load_data
from src.models.model_2.model.pipelines_2h import pipeline

train_data, augmented_data, test_data = load_data('2_00h')

all_train_data_transformed = pipeline.fit_transform(pd.concat([train_data, augmente

X_train, y_train = all_train_data_transformed.iloc[len(train_data):].drop(columns=[
X_augmented, y_augmented = all_train_data_transformed.iloc[:len(train_data)].drop(c
```

Lazy Predict

To gain quick insights into the performance of different regression models, we can use the `LazyPredict` library. This library automates the training and evaluation of a variety of regression models on the data, providing a fast and comprehensive overview of model performance. For our use case, we developed a custom wrapper around `LazyPredict` to have more control over the models applied.

```
from src.features.helpers.LazyPredict import get_lazy_regressor
from sklearn.model_selection import train_test_split

X_augmented_train, X_augmented_test, y_augmented_train, y_augmented_test = train_te

X_train_all = pd.concat([X_train, X_augmented_train], axis=0)
y_train_all = pd.concat([y_train, y_augmented_train], axis=0)

lazy_predict_regressor = get_lazy_regressor(exclude=['SVN'])
models, predictions = lazy_predict_regressor.fit(X_train=X_train_all, y_train=y_tra
models
```

Model Selection

Based on the results from the `LazyPredict` library, we can identify and select the best performing models from different categories, fine-tune and combine them into a `StackingRegressor` for the final prediction.

In this case we will use:

- `HistGradientBoostingRegressor`
- `LassoLarsICRegressor`
- `KNNRegressor`
- `XGBRegressor`

Why HistGradientBoostingRegressor

Category: Gradient Boosting Model

Strengths:

- Efficient implementation of gradient boosting, optimized for large datasets with categorical features.

- Handles missing data well and works effectively on high-dimensional datasets.
- Generally robust to overfitting due to regularization.

Unique Contribution to Stacking:

- Captures complex, non-linear relationships.
- Performs well in terms of accuracy on structured data and integrates nicely with other weaker models.

Why LassoLarsICRegressor

Category: Linear Model

Strengths:

- A regression model that combines Lasso (L1 regularization) with a model selection method based on the Akaike Information Criterion (AIC) or Bayes Information Criterion (BIC).
- Particularly effective for datasets with a large number of features but where most coefficients are zero (sparse datasets).

Unique Contribution to Stacking:

- Provides a strong linear baseline that helps the ensemble learn from both linear trends and more complex patterns captured by non-linear models.
- Avoids overfitting by feature selection.

Why KNNRegressor

Category: Instance-Based Learning (Non-parametric)

Strengths:

- Simple yet effective for small-to-medium datasets, particularly when the relationship between features and the target variable is highly localized.
- Handles non-linear relationships without assuming any prior distribution.

Unique Contribution to Stacking:

- Introduces local prediction capability that complements global models like gradient boosting.
- Offers diversity to the ensemble, as its predictions are based purely on similarity rather than a parametric model.

Why XGBRegressor

Category: Gradient Boosting Model

Strengths:

- High performance and efficiency due to optimized implementations.
- Supports a variety of tuning options and regularization techniques (L1 and L2).
- Often achieves state-of-the-art results in many regression tasks.

Unique Contribution to Stacking:

- Brings additional predictive power, especially when the dataset has complex feature interactions.
- Complements HistGradientBoostingRegressor by leveraging different gradient boosting frameworks.

Why Stacking these models

1. Diversity: Each model belongs to a different category (linear, boosting, instance-based), ensuring diverse perspectives.
2. Complementary Strengths: Combining models that excel in different aspects (e.g., linear vs. non-linear, global vs. local) leads to a well-rounded regressor.
3. Error Reduction: Aggregating predictions mitigates individual model weaknesses, reducing bias and variance.
4. Improved Generalization: Stacking effectively captures patterns that individual models may miss, enhancing overall performance.

By carefully tuning these models and stacking their predictions, the ensemble benefits from their combined strengths, yielding a robust and accurate final regressor.

Feature Selection

```
from src.features.helpers.load_data import load_data
from src.models.model_2.model.pipelines_2h import pipeline
import pandas as pd

train_data, augmented_data, test_data = load_data('2_00h')

all_train_data_transformed = pipeline.fit_transform(pd.concat([train_data, augmente
X_train, y_train = all_train_data_transformed.iloc[len(train_data):].drop(columns=[
```

Feature Importance

For each of the models selected in the previous section, we calculate feature importance to identify the most important features for predictions.

HistGradientBoostingRegressor

```
import pandas as pd
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.inspection import permutation_importance

model = HistGradientBoostingRegressor()
model.fit(X_train, y_train)
importances = permutation_importance(model, X_train, y_train, n_repeats=10, random_
hgb_feature_importances = pd.Series(importances.importances_mean, index=X_train.col
hgb_feature_importances = hgb_feature_importances / hgb_feature_importances.sum()
mask = ~(hgb_feature_importances.index.str.startswith('hour') | hgb_feature_importa
hgb_feature_importances = hgb_feature_importances[mask]
hgb_feature_importances = pd.DataFrame(hgb_feature_importances, columns=['hgb_impor
```

LassoLarsICRegressor

```
from sklearn.linear_model import LassoLarsIC

model = LassoLarsIC()
model.fit(X_train, y_train)
lasso_feature_importances = pd.Series(abs(model.coef_), index=X_train.columns)
lasso_feature_importances = lasso_feature_importances / lasso_feature_importances.s
mask = ~(lasso_feature_importances.index.str.startswith('hour') | lasso_feature_imp
lasso_feature_importances = lasso_feature_importances[mask]
lasso_feature_importances = pd.DataFrame(lasso_feature_importances, columns=['lasso
```

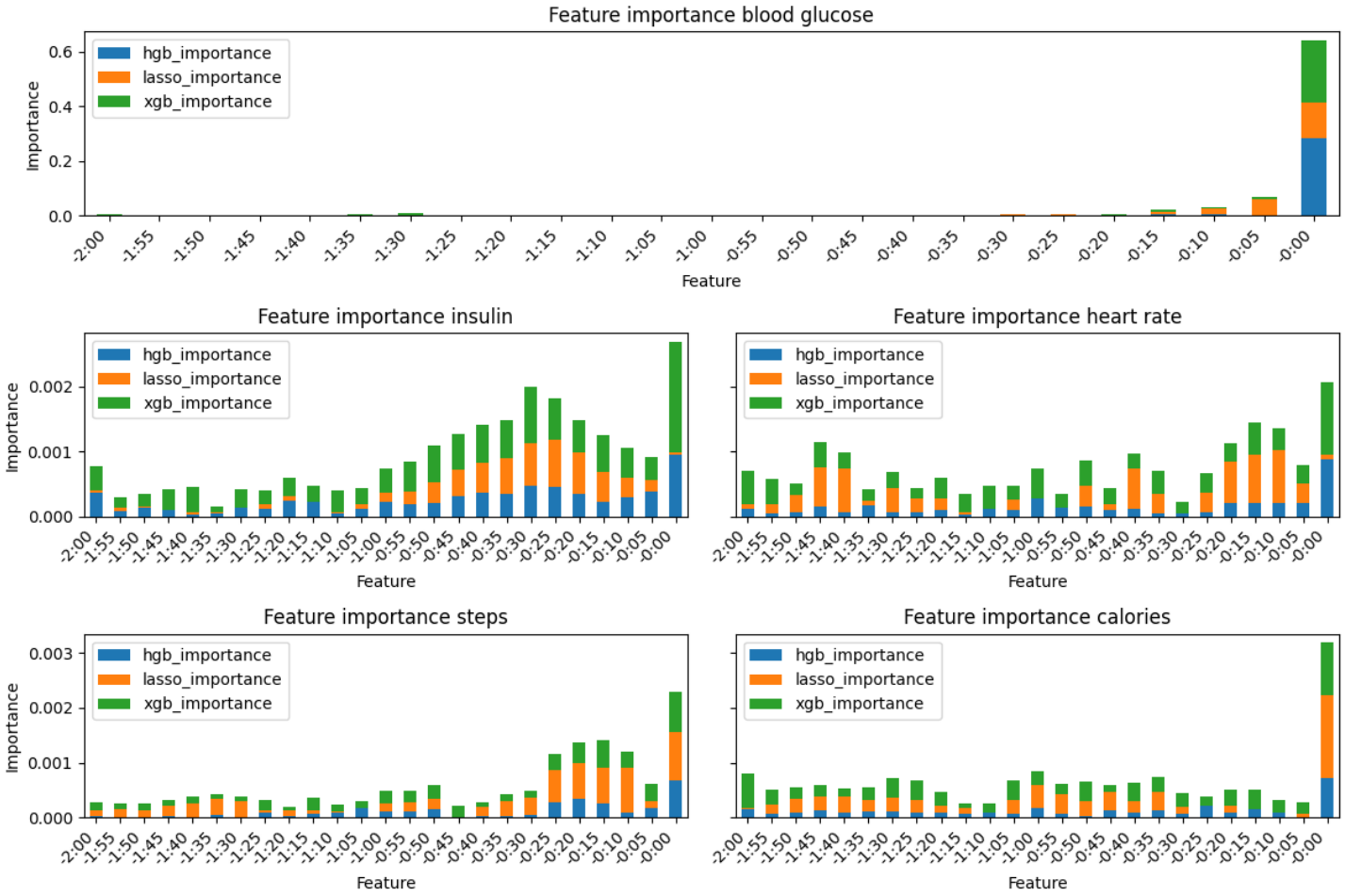
XGBRegressor

```
from xgboost import XGBRegressor
from sklearn.inspection import permutation_importance

model = XGBRegressor()
model.fit(X_train, y_train)
importances = permutation_importance(model, X_train, y_train, n_repeats=10, random_
xgb_feature_importances = pd.Series(importances.importances_mean, index=X_train.col
xgb_feature_importances = xgb_feature_importances / xgb_feature_importances.sum()
mask = ~(xgb_feature_importances.index.str.startswith('hour') | xgb_feature_importa
xgb_feature_importances = xgb_feature_importances[mask]
xgb_feature_importances = pd.DataFrame(xgb_feature_importances, columns=['xgb_impor
```

Plotting Feature Importance by Parameters

By summing and averaging the feature importance scores from the three models, we can visualize the most influential features across models.

```
from helpers import plot_feature_importance_chart
%matplotlib inline

feature_importances = pd.concat([hgb_feature_importances, lasso_feature_importances
feature_importances = feature_importances / len(feature_importances.columns)

plot_feature_importance_chart(feature_importances)
```



The chart shows, that the most important features are the blood glucose levels until -0:30h, the insulin levels until -1:00h, the heart rate until -1:00h, the steps until -1:00h and the calories until -1:00h.

Feature Selection

For the modelling section we will choose the following features:

- bg-0:00 - bg-1:00
- insulin-0:00 - insulin-1:00
- hr-0:00 - hr-1:00
- steps-0:00 - steps-1:00
- cals-0:00 - cals-1:00

Hyperparameter Tuning

For each of previously selected models, we will perform hyperparameter tuning using the **Bayesian Optimization** method. This approach efficiently searches the hyperparameter space to identify the optimal configuration for each model.

Load Data

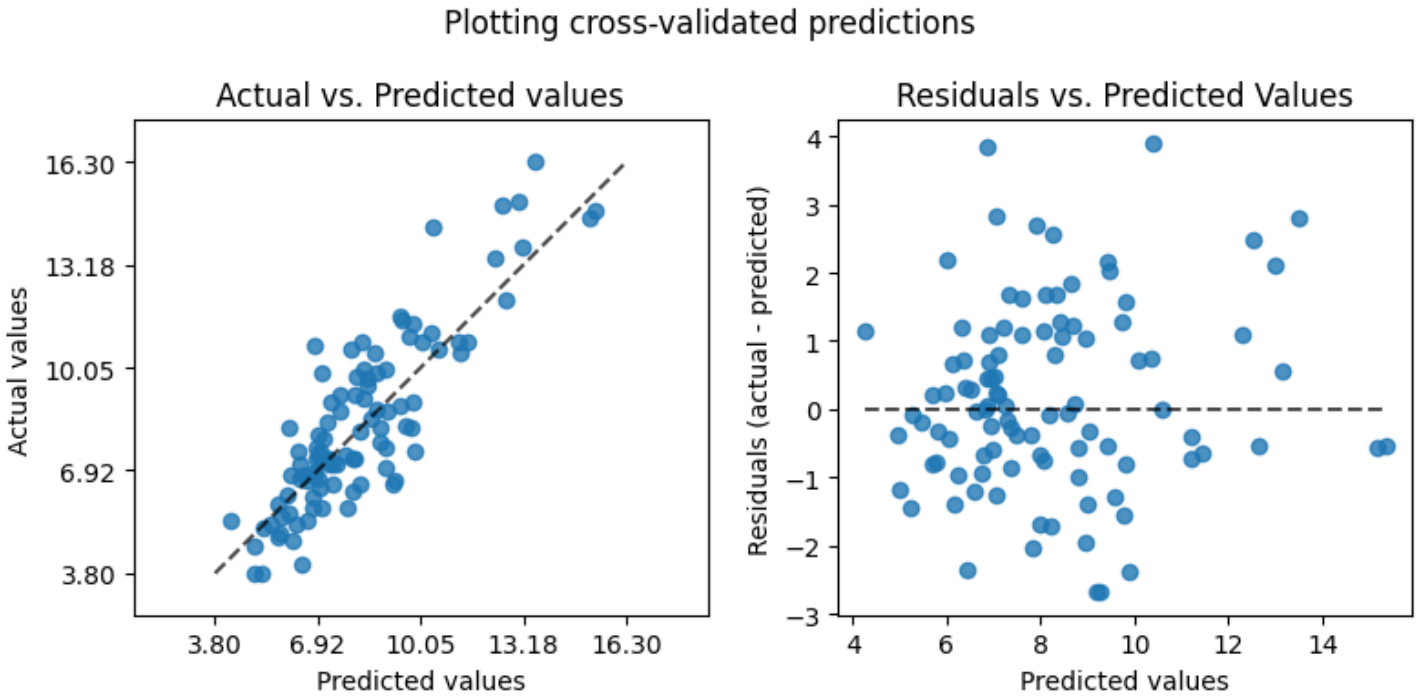
```
import warnings

warnings.filterwarnings('ignore')
```

```
from src.features.helpers.load_data import load_data
from src.models.model_2.model.pipelines import pipeline
import pandas as pd

train_data, augmented_data, test_data = load_data('1_00h')

all_train_data_transformed = pipeline.fit_transform(pd.concat([train_data, augmente

X_train = all_train_data_transformed.iloc[:len(train_data)].drop(columns=['bg+1:00'
y_train = all_train_data_transformed.iloc[:len(train_data)]['bg+1:00']

X_augmented = all_train_data_transformed.iloc[len(train_data):].drop(columns=['bg+1
y_augmented = all_train_data_transformed.iloc[len(train_data):]['bg+1:00']
```

HistGradientBoostingRegressor Hyperparameter Tuning

```
from sklearn.ensemble import HistGradientBoostingRegressor
from skopt.space import Real, Integer
from helpers import tune_hyperparameters

search_space = {
    'learning_rate': Real(0.005, 0.3, prior='log-uniform'),
    'max_iter': Integer(100, 1500),
    'max_depth': Integer(6, 8),
    'min_samples_leaf': Integer(15, 30),
    'max_leaf_nodes': Integer(10, 30),
    'l2_regularization': Real(1e-4, 1e-2, prior='log-uniform'),
    'n_iter_no_change': Integer(10, 15),
    'validation_fraction': Real(0.1, 0.3),
}

model = HistGradientBoostingRegressor(
    scoring='neg_mean_squared_error',
    max_bins=255,
    random_state=42
)

_, best_params = tune_hyperparameters(
    model, search_space,
    X_train, y_train,
    X_augmented, y_augmented,
    num_iter=50, n_splits=5,
    verbose=False, show_results=True
)

pd.DataFrame(best_params, index=[0]).T
```

```
RMSE: 1.4611029397279351
R2 Score: 0.7790285485484115
```

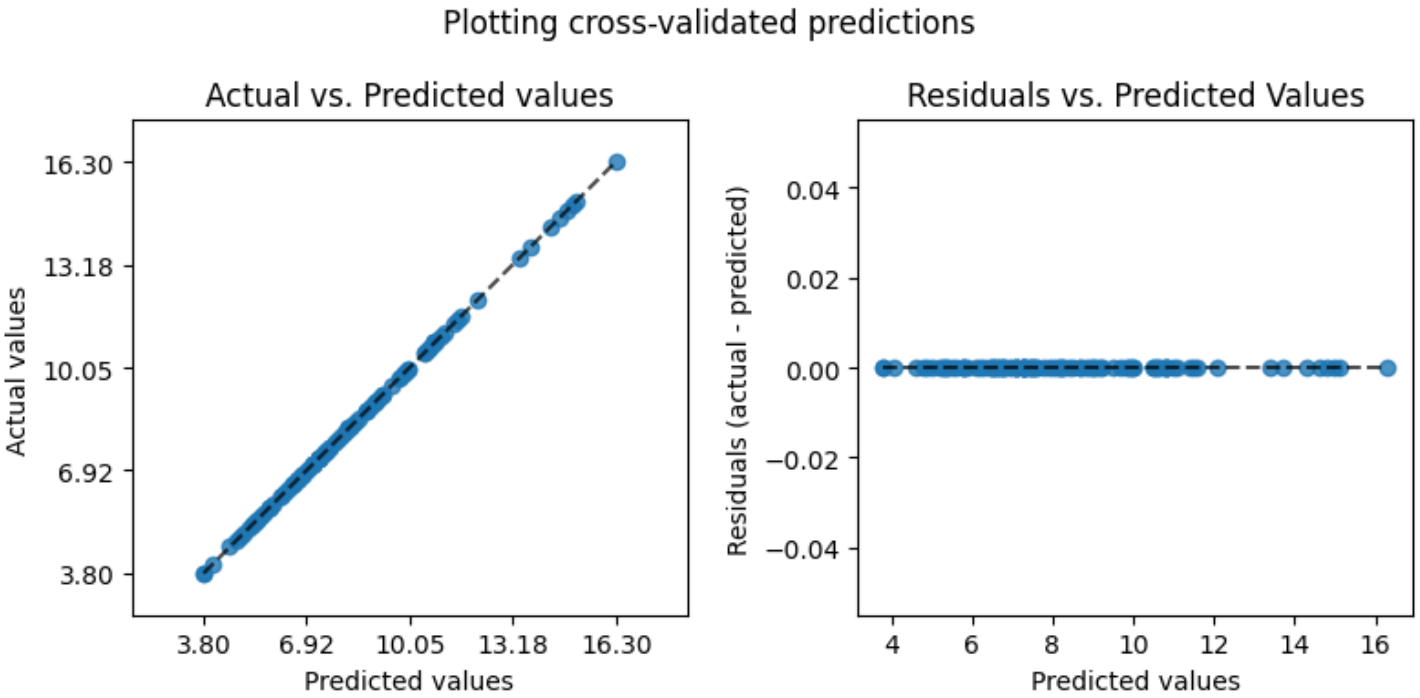


	0
l2_regularization	0.000100
learning_rate	0.166051
max_depth	8.000000
max_iter	1500.000000
max_leaf_nodes	30.000000
min_samples_leaf	15.000000
n_iter_no_change	15.000000
validation_fraction	0.100000

KNeighborsRegressor Hyperparameter Tuning

```
from sklearn.neighbors import KNeighborsRegressor
from skopt.space import Integer, Categorical
from helpers import tune_hyperparameters

search_space = {
    'n_neighbors': Integer(3, 15),
    'weights': Categorical(['uniform', 'distance']),
    'p': Categorical([1, 2]),
    'metric': Categorical(['minkowski', 'euclidean', 'chebyshev']),
    'leaf_size': Integer(10, 50),
}

model = KNeighborsRegressor(n_jobs=-1)
_, best_params = tune_hyperparameters(
    model, search_space,
    X_train, y_train,
    X_augmented, y_augmented,
    num_iter=50, n_splits=5,
    verbose=False, show_results=True
)

pd.DataFrame(best_params, index=[0]).T
```

```
RMSE: 0.06926307319532808
R2 Score: 0.9995034327722873
```

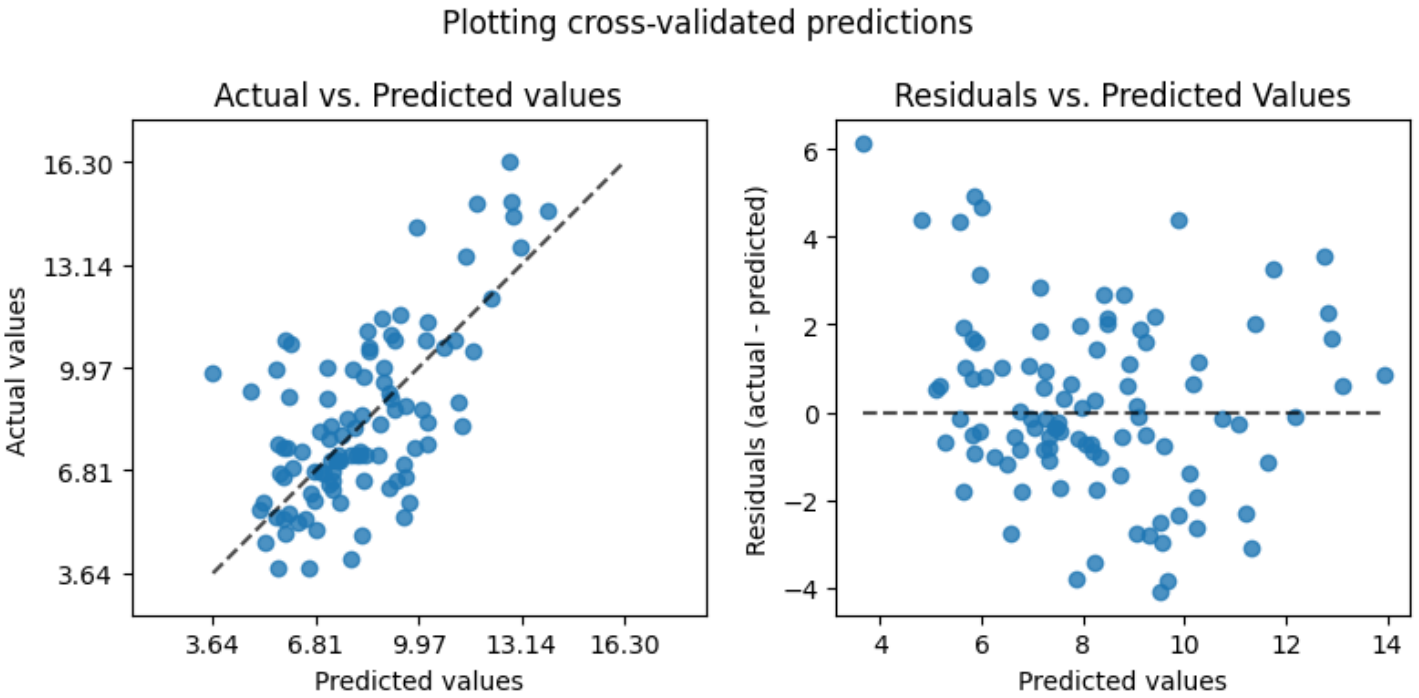


	0
leaf_size	10
metric	minkowski
n_neighbors	3
p	1
weights	distance

LassoLarsICRegressor Hyperparameter Tuning

```
from sklearn.linear_model import LassoLarsIC
from skopt.space import Categorical, Real, Integer
from helpers import tune_hyperparameters

search_space = {
    'criterion': Categorical(['aic', 'bic']),
    'eps': Real(1e-6, 1e-1, prior='log-uniform'),
    'max_iter': Integer(1000, 10000),
    'noise_variance': Real(1e-6, 1e-2, prior='log-uniform'),
    'positive': Categorical([True, False]),
}

model = LassoLarsIC()
_, best_params = tune_hyperparameters(
    model, search_space,
    X_train, y_train,
    X_augmented, y_augmented,
    num_iter=50, n_splits=5,
    verbose=False, show_results=True
)

pd.DataFrame(best_params, index=[0]).T
```

```
RMSE: 2.13674986642704
R2 Score: 0.5274128483110203
```

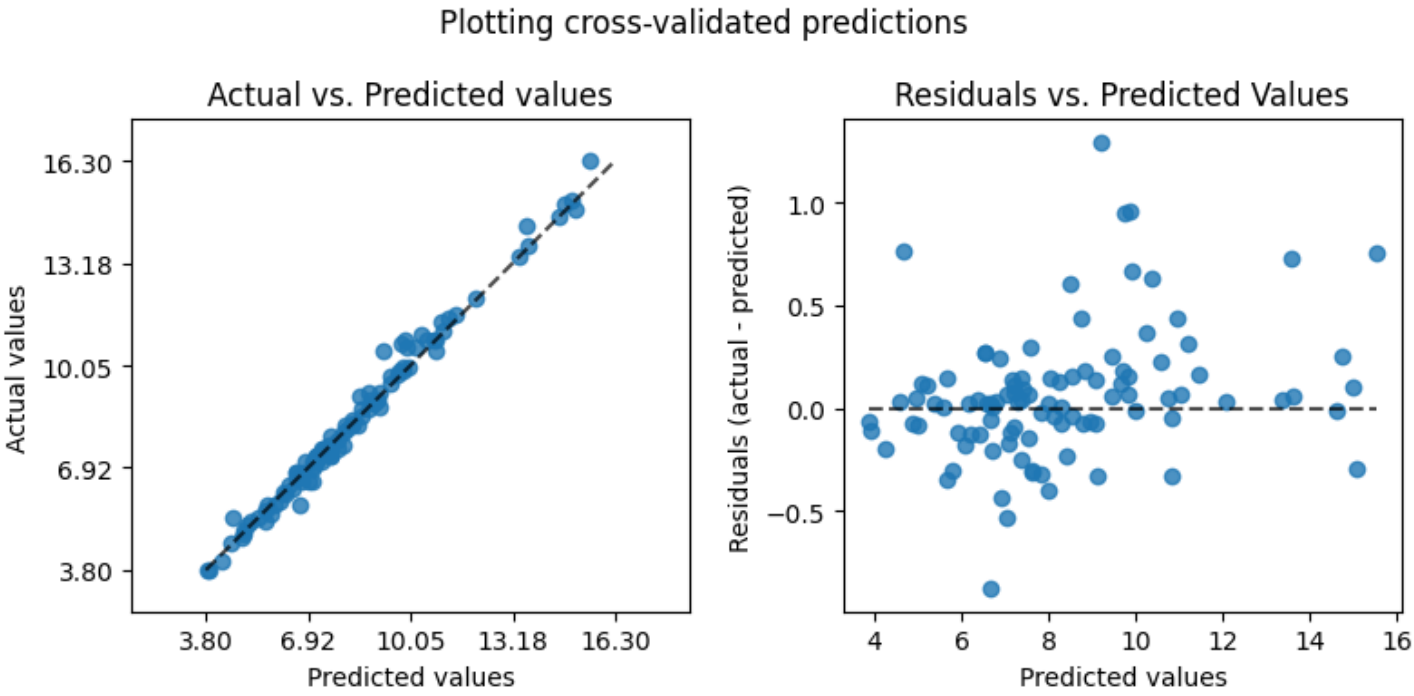


	0
criterion	aic
eps	0.039229
max_iter	1944
noise_variance	0.000054
positive	False

XGBRegressor Hyperparameter Tuning

```
from xgboost import XGBRegressor
from skopt.space import Integer, Real
from helpers import tune_hyperparameters

search_space = {
    'n_estimators': Integer(1500, 5000),
    'learning_rate': Real(0.01, 0.05, 'log-uniform'),
    'max_depth': Integer(3, 10),
    'min_child_weight': Integer(2, 7),
    'subsample': Real(0.5, 0.8),
    'colsample_bytree': Real(0.6, 0.9),
    'gamma': Real(0.1, 2, 'log-uniform'),
    'reg_alpha': Real(1, 50, 'log-uniform'),
    'reg_lambda': Real(1, 20, 'log-uniform'),
}
model = XGBRegressor(random_state=42, n_jobs=-1)
_, best_params = tune_hyperparameters(
    model, search_space,
    X_train, y_train,
    X_augmented, y_augmented,
    num_iter=50, n_splits=5,
    verbose=False, show_results=True
)

pd.DataFrame(best_params, index=[0]).T
```

```
RMSE: 0.3157643471811322
R2 Score: 0.9896795003341667
```



	0
colsample_bytree	0.900000
gamma	0.100000
learning_rate	0.050000
max_depth	10.000000
min_child_weight	7.000000
n_estimators	5000.000000
reg_alpha	4.946715
reg_lambda	1.000000
subsample	0.800000

Modelling

In this notebook, we will train and fine-tune the models selected in the previous notebook using the prepared dataset and the optimized hyperparameters. Following models will be trained and tuned:

- HistGradientBoostingRegressor
- KNeighborsRegressor
- LassoLarsICRegressor
- XGBRegressor

The goal is to combine these models into a StackingRegressor to improve the predictive performance. We will use the **Ridge** model as the final estimator.

Load Data

```
import warnings

warnings.filterwarnings('ignore')
```

```
from sklearn.model_selection import train_test_split
from src.features.helpers.load_data import load_data
from src.models.model_2.model.pipelines import pipeline
import pandas as pd

train_data, augmented_data, test_data = load_data('1_00h')

all_train_data_transformed = pipeline.fit_transform(pd.concat([train_data, augmente

X_train = all_train_data_transformed.iloc[:len(train_data)].drop(columns=['bg+1:00'
y_train = all_train_data_transformed.iloc[:len(train_data)]['bg+1:00']

X_augmented = all_train_data_transformed.iloc[len(train_data):].drop(columns=['bg+1
y_augmented = all_train_data_transformed.iloc[len(train_data):]['bg+1:00']

X_augmented_train, X_augmented_val, y_augmented_train, y_augmented_val = train_test

X_train = pd.concat([X_train, X_augmented_train])
y_train = pd.concat([y_train, y_augmented_train])

X_test = pipeline.transform(test_data)
```

Model Setup

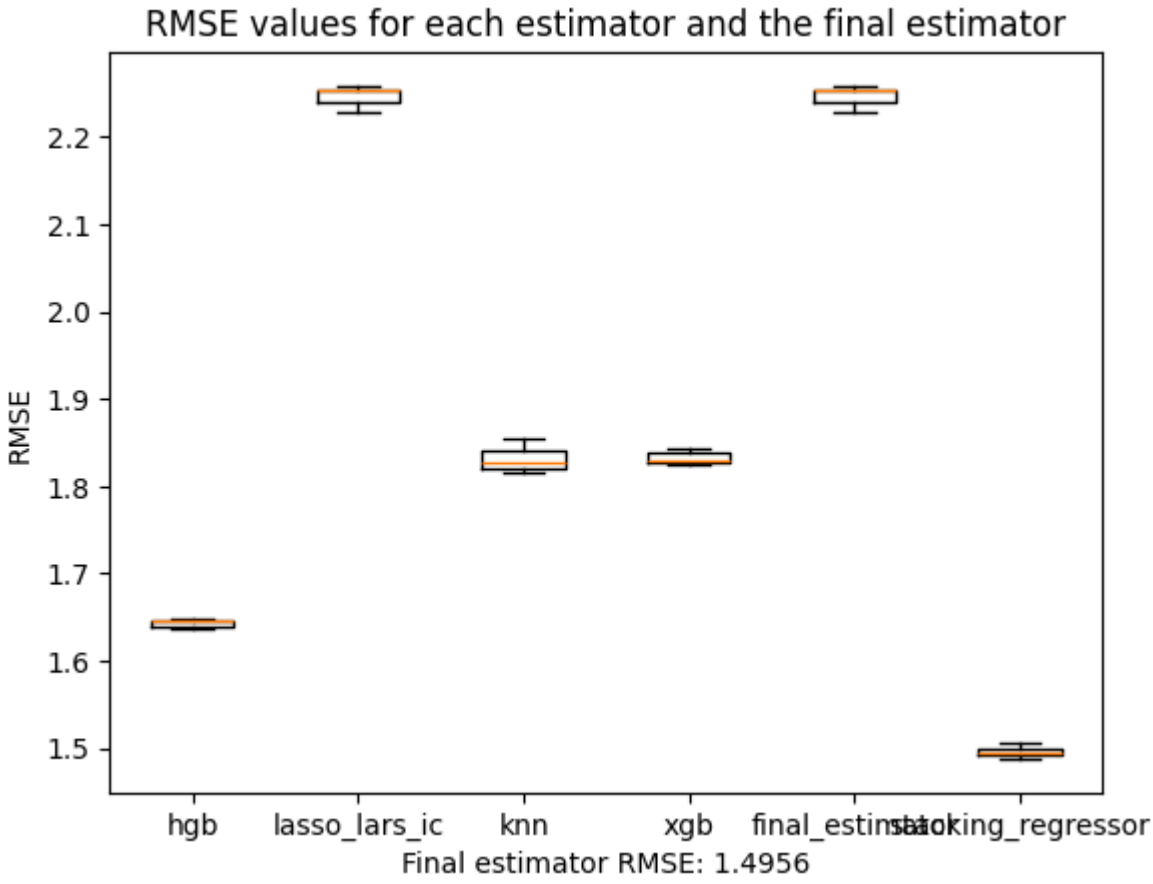
```
from xgboost import XGBRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LassoLarsIC, Ridge
from sklearn.ensemble import HistGradientBoostingRegressor, StackingRegressor

hgb_base_estimator = HistGradientBoostingRegressor(
    l2_regularization=0.0001,
    learning_rate=0.1,
    max_depth=11,
    max_iter=1500,
    min_samples_leaf=10,
    n_iter_no_change=22,
    random_state=42,
)

lasso_lars_ic_base_estimator = LassoLarsIC(
    criterion='aic',
    eps=0.03922948513965659,
    max_iter=1944,
    noise_variance=5.4116687755186035e-05,
    positive=False,
)

knn_base_estimator = KNeighborsRegressor(
    leaf_size=30,
    metric='minkowski',
    n_neighbors=7,
    p=2,
    weights='uniform'
)

xgb_base_estimator = XGBRegressor(
    n_estimators=1000,
    max_depth=6,
    learning_rate=0.05,
    min_child_weight=6,
    colsample_bytree=0.8,
    subsample=0.8,
    gamma=1,
    reg_lambda=10.0,
    reg_alpha=1.0,
    objective='reg:squarederror',
    random_state=42,
)

estimators = [
    ('hgb', hgb_base_estimator),
    ('lasso_lars_ic', lasso_lars_ic_base_estimator),
    ('knn', knn_base_estimator),
    ('xgb', xgb_base_estimator),
]

model = StackingRegressor(
```

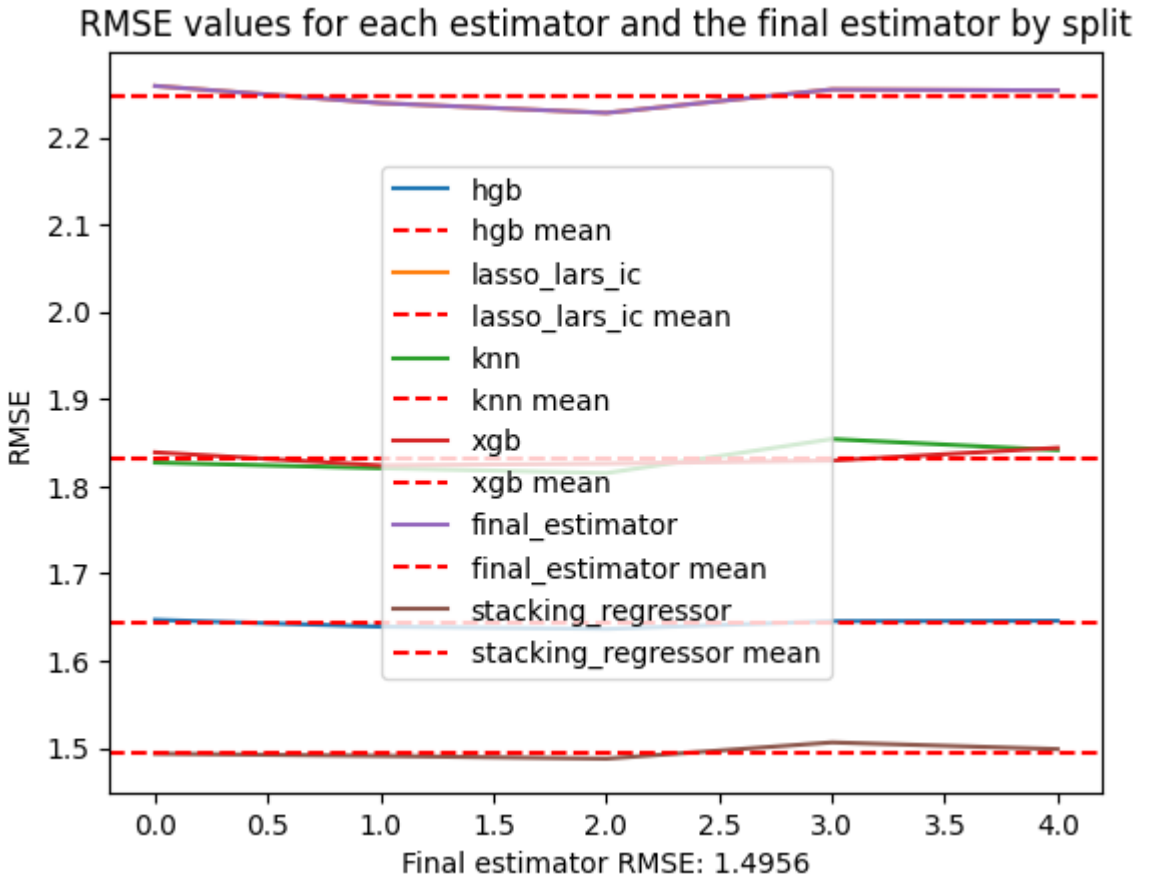
```
    estimators=estimators,
    final_estimator=Ridge(alpha=0.1), n_jobs=-1, verbose=0
)
```

Performance Calculations with Cross Validation

```
from datetime import datetime
from model_performance_calculations import calculate_stacking_regressor_performance

date_time = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
model_name = f'{date_time}-baseline-model'

performances = calculate_stacking_regressor_performance(model, X_train, y_train, X_

get_rmse_boxplot_chart(performances).show()
get_rmse_line_chart(performances).show()
```

```
Final estimator RMSE: 1.4956417733797829
Final estimator R2: 0.7847963155567121
Final estimator MSE: 2.2369870974974537
Final estimator MAE: 1.1116463866250015
```


RMSE values for each estimator and the final estimator
RMSE
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
hgb
lasso_lars_ic
knn
xgb
final_estimator
stacking_regressor
Final estimator RMSE: 1.4956


RMSE values for each estimator and the final estimator by split
RMSE
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
hgb
hgb mean
lasso_lars_ic
lasso_lars_ic mean
knn
knn mean
xgb
xgb mean
final_estimator
final_estimator mean
stacking_regressor
stacking_regressor mean
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Final estimator RMSE: 1.4956

Train the model with all data and predict for test data

```
import numpy as np

model.fit(pd.concat([X_train, X_augmented_train]), pd.concat([y_train, y_augmented_
y_pred = model.predict(X_test)

# do a sanity check, we should not have negative values
if np.sum(y_pred < 0) > 0:
    print(f'Number of negative values: {np.sum(y_pred < 0)}')
    bg_min_train = np.min(y_train)
    print(f'Min value: {np.min(y_pred)}')
    y_pred = y_pred.apply(lambda x: bg_min_train if x < 0 else x)

test_data['bg+1:00'] = y_pred
```

Save the predictions for submission

```
import os

submission = pd.DataFrame(test_data['bg+1:00'])
submission.to_csv(f'submission-{os.path.basename(os.getcwd())}.csv')
submission
```

	bg+1:00
id	
p01_8459	8.756893
p01_8460	5.118244
p01_8461	7.710794
p01_8462	11.345586
p01_8463	6.470444
...	...
p24_256	5.809733
p24_257	8.780385
p24_258	5.700356
p24_259	7.914720
p24_260	5.644072

3644 rows × 1 columns

Conclusions

In this chapter, we described and explained another approach to predicting blood glucose levels one hour ahead using participant data with lag features. The approach involved extensive data exploration, feature engineering, and model tuning to achieve optimal performance.

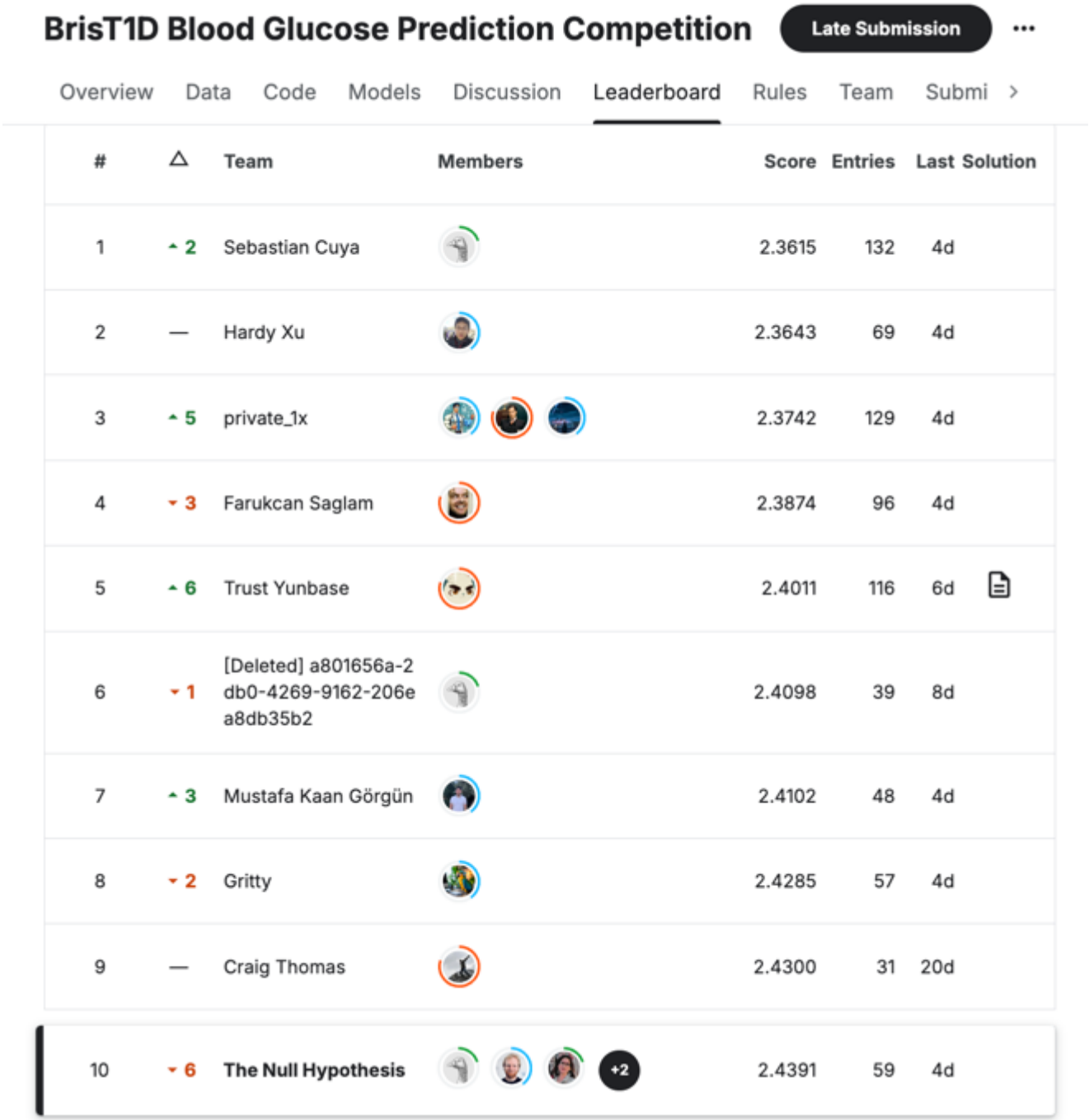
Key Highlights:

1. Data Augmentation:
    - Unlike the first approach, where only the original training data was used, we augmented the training data by shifting the test data by a minimum of 1 hour and a maximum of 4 hours to generate additional samples. This strategy increased both the quantity and quality of training data.
2. Feature Engineering and Preprocessing:
    - We explored the dataset carefully, identified the most important features, and engineered features to enhance model performance.

3. Model Evaluation and Selection:
    - Using LazyPredict, we evaluated multiple models and identified the top-performing ones for further fine-tuning.
    - We developed a set of custom HyperParameterTuner classes for the best performing models identified by LazyPredict, which automatically perform Bayesian Optimization (BayesSearchCV) and a 5-fold cross-validation to identify an optimal set of hyper-parameters.
    - Hyperparameter tuning was conducted using BayesSearchCV (Bayesian Optimization) and 5-fold cross-validation to identify an optimal set of hyper-parameters.
    - Then, we fine-tuned the models using the optimal hyper-parameters, evaluated their performance and selected the best performing models from different categories.
    - Based on the selected models, we could determine the most important features, which are within the last 1 hour of data. Having this information, we could adjust our data augmentation to further improve the quantity and quality of data.
4. Stacking Ensemble:
    - The best-performing models from different categories were combined into a `StackingRegressor` ensemble model with `Ridge` as the meta-learner.
    - We evaluated the ensemble model and showed that StackingRegressor outperformed individual models, demonstrating the benefits of combining diverse algorithms.
5. Final Model Performance:
    - The final model predicted blood glucose levels one hour ahead using six hours of participant data.
    - Submissions to the Kaggle competition were evaluated based on Root Mean Square Error (RMSE) between the predicted blood glucose levels and the actual values.
    - Our model achieved an RMSE of **2.4224** on the test data.

Kaggle Results:

- Public Leaderboard Rank: 4
- Private Leaderboard Rank: 10



> In conclusion, we successfully developed a remarkable model that predicts blood glucose levels one hour ahead with a high degree of accuracy. This predictive capability can significantly aid individuals with type 1 diabetes in managing their condition more effectively.

Summary and Final Thoughts

In this report, we presented our approaches to predicting blood glucose levels one hour ahead using six hours of participant data. Our work involved a comprehensive workflow, including data exploration, preprocessing, feature engineering as well as model training. Furthermore, a custom cross-validation strategy was implemented to evaluate model performance and optimize its hyperparameters.

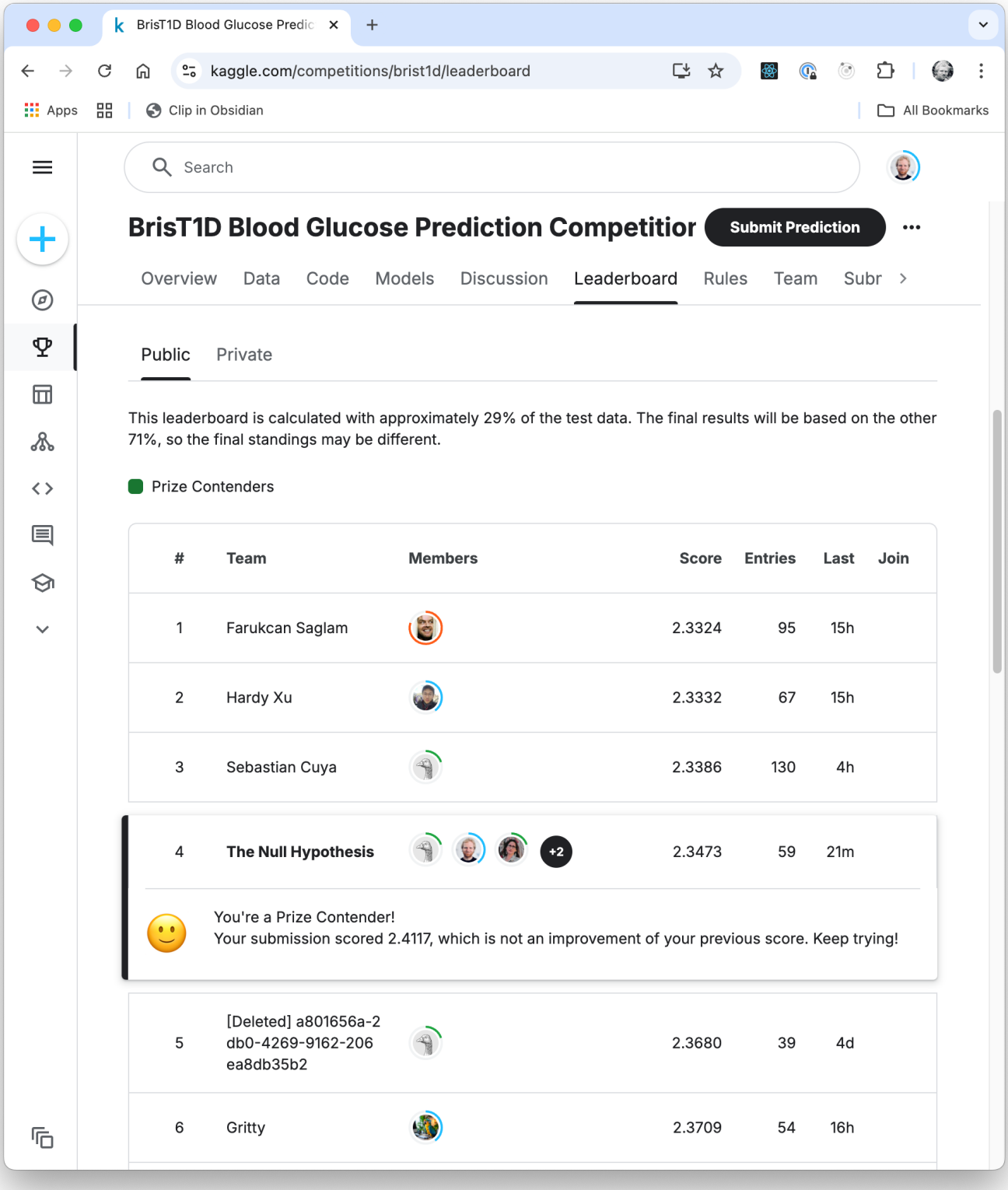
Two Modelling Approaches

We developed two different modelling approaches. The first approach utilized the training data as provided, while the second one augmented the training dataset by generating additional data from the test data, shifting it by one to four hours. This data augmentation strategy significantly improved the model's performance by providing additional training examples.

Our continious trials with different types of models achieved a Kaggle Score of 2.45 on the public leaderboard, which was a significant improvement over the baseline model. So we kept further validations using a custom cross-validation strategy, which showed consistent results across different folds.

Results

Our best model achieved a Kaggle Score of **2.3472** on the public leaderboard, which looked very promising. Post-competition evaluation on the private leaderboard, assessing 80% of the unseen test data, yielded an RMSE of **2.4391**, which means that our model was able to predict 20% of the test data with this RMSE. Based on this evaluations, our model was ranked 10th in the competition.


BrisT1D Blood Glucose Predic
kaggle.com/competitions/brist1d/leaderboard
Apps
Clip in Obsidian
All Bookmarks
Search
BrisT1D Blood Glucose Prediction Competitior
Submit Prediction
Overview
Data
Code
Models
Discussion
Leaderboard
Rules
Team
Subr
Public
Private
This leaderboard is calculated with approximately 29% of the test data. The final results will be based on the other 71%, so the final standings may be different.
Prize Contenders
Team Members Score Entries Last Join
1 Farukcan Saglam 2.3324 95 15h
2 Hardy Xu 2.3332 67 15h
3 Sebastian Cuya 2.3386 130 4h
4 The Null Hypothesis +2 2.3473 59 21m
You're a Prize Contender!
Your submission scored 2.4117, which is not an improvement of your previous score. Keep trying!
5 [Deleted] a801656a-2db0-4269-9162-206ea8db35b2 2.3680 39 4d
6 Gritty 2.3709 54 16h

Challenges

We encountered several challenges during the development of our models, which we tried to address through various strategies.

Imbalanced Training and Test Data

One of the main issues was the imbalance between training and test data, as some patients in the test set were absent from the training set. This made predictions for these patients particularly challenging. To address this, we leveraged data augmentation techniques to create additional training samples derived from the test data.

Kaggle RMSE and Local RMSE Discrepancy

Another big challenge was the discrepancy between local RMSE scores and Kaggle leaderboard scores. Despite implementing various strategies such as feature selection, hyperparameter tuning, and refined cross-validation, the gap persisted. We have different hypotheses on why this discrepancy exists, which have to be proven in future work.

- The number of features is too high and introduces noise into the model.
- The model is overfitting the training data (Trees too deep, KNeighbors too low) and not generalizing well to the test data on Kaggle.
- The model is not able to capture the underlying patterns in the data.

Future Directions

This project successfully demonstrated the utility of data augmentation and ensemble modelling. To enhance the model's performance and generalization, the following areas could be explored deeper in the future:

- **Feature Reduction**: We could explore further feature reduction techniques to identify the most important features and reduce the dimensionality of the dataset.
- **Noise Reduction**: We could explore techniques to reduce noise in the data and improve the quality of the predictions.

- **Cross-Validation**: We could explore different cross-validation techniques which reflect better the Kaggle score.
- **Deep Learning Approaches:** Although our attempt to integrate a basic Deep Neural Network (DNN) into the Stacking Ensemble model did not yield significant improvements, more sophisticated deep learning architectures and strategies could be explored in the future to unlock additional potential.

By addressing these areas, the model could achieve even greater accuracy and reliability, providing more value to users and competitive performance in future challenges.

Final Thoughts

This project has been an incredible learning experience for our team. We have gained valuable insights into data preprocessing, feature engineering, model development, and performance evaluation. We also learned how to work collaboratively on such a project, communicate effectively, and solve problems as a team. We applied various machine learning techniques as well as integrating them into a robust ensemble model like stacking.

Our solution was ranked in the **top 1% of the Kaggle public Leaderboard**, which is a great achievement for us and a reflektion of the hard work we invested in this project. This achievement is something we are genuinely proud of, as it reflects our effort of out team. We believe that the skills and knowledge we have developed during this project will be valuable in our future careers as data scientists and machine learning engineers.

Appendix

The appendix provides additional information that supports the main content of the report. This section includes:

- Transformer Classes: Custom transformer classes used for data preprocessing and feature engineering.
- Hyperparameter Tuner Classes: Custom hyperparameter tuner classes used for model optimization.
- Custom Splitter Class: A custom splitter class used for cross-validation in the hyperparameter optimization process.

Transformer classes

Transforming Date and Time Features

DateTimeHourTransformer

The `DateTimeHourTransformer` has two different modes of operation.

- `sin_cos`, which converts the hour of the day into two features, `hour_sin` and `hour_cos`, which represent the hour of the day as a sine and cosine wave.
- `bins`, which converts the hour of the day into a categorical feature with a specified number of bins.

```python
from typing import Literal

import numpy as np
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin


class DateTimeHourTransformer(BaseEstimator, TransformerMixin):
  _result_columns: list[str]
  _source_time_format: str
  _time_column: str
  _type: Literal['sin_cos', 'bins']

  def __init__(self, time_column: str, result_column: str, type: Literal['sin_cos',
               source_time_format: str = '%H:%M:%S'):
    self._time_column = time_column
    self._drop_time_column = drop_time_column
    self._result_column = result_column
    self._type = type
    self._number_of_bins = number_of_bins
    self._source_time_format = source_time_format

  def fit(self, X: pd.DataFrame, y=None):
    return self

  def transform(self, X: pd.DataFrame):
    X = X.copy()

    if not self._time_column in X.columns:
      raise ValueError('time_column must be set')

    if self._type == 'sin_cos':
      hour_values = pd.to_datetime(X[self._time_column], format=self._source_time_f
      minute_values = pd.to_datetime(X[self._time_column], format=self._source_time_

      X[f'{self._result_column}_sin'] = np.sin(2 * np.pi * (hour_values + minute_va
      X[f'{self._result_column}_cos'] = np.cos(2 * np.pi * (hour_values + minute_va
      columns = list(X.columns)
      columns.remove(f'{self._result_column}_sin')
      columns.remove(f'{self._result_column}_cos')
      columns.insert(columns.index(self._time_column) + 1, f'{self._result_column}_
      columns.insert(columns.index(self._time_column) + 2, f'{self._result_column}_
      X = X[columns]

    elif self._type == 'bins':
      time_bins = np.linspace(0, 24, self._number_of_bins + 1)
      hour_values = pd.to_datetime(X[self._time_column], format=self._source_time_f
      X[self._result_column] = pd.cut(hour_values, bins=time_bins, labels=range(sel
      columns = list(X.columns)
      columns.remove(self._result_column)
      columns.insert(columns.index(self._time_column) + 1, self._result_column)
      X = X[columns]
```

```
    if self._drop_time_column:
      X = X.drop(columns=[self._time_column])

    return X
```

DayPhaseTransformer

The `DayPhaseTransformer` converts the hour of the day into a categorical feature representing 6 phases of the day:

- `morning` - 6 to 9
- `noon` - 10 to 13
- `afternoon` - 14 to 17
- `evening` - 18 to 21
- `late_evening` - 22 to 24
- `night` - 0 to 5

```
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin


class DayPhaseTransformer(BaseEstimator, TransformerMixin):
  _result_column: str
  _time_column: str

  def __init__(self, time_column: str, time_format: str, result_column: str, drop_t
    self._time_column = time_column
    self._time_format = time_format
    self._result_column = result_column
    self._drop_time_column = drop_time_column
    self._ignore_errors = ignore_errors

  def _get_day_phase(self, hour: int):
    if 6 <= hour <= 9:
      return 'morning'
    elif 10 <= hour <= 13:
      return 'noon'
    elif 14 <= hour <= 17:
      return 'afternoon'
    elif 18 <= hour <= 21:
      return 'evening'
    elif 22 <= hour <= 24:
      return 'late_evening'
    else:
      return 'night'

  def fit(self, X: pd.DataFrame, y=None):
    return self

  def transform(self, X: pd.DataFrame):
    X = X.copy()
    if not self._time_column in X.columns:
      if self._ignore_errors:
        return X
      raise ValueError('time_column must be set')

    X[self._result_column] = pd.to_datetime(X[self._time_column], format=self._time_

    # reorder result column directly after time column
    columns = list(X.columns)
    columns.remove(self._result_column)
    columns.insert(columns.index(self._time_column) + 1, self._result_column)
    X = X[columns]

    if self._drop_time_column:
      X = X.drop(columns=[self._time_column])

    return X
```

DropColumnsTransformer

The `DropColumnsTransformer` removes columns from a DataFrame based on a list of column names or columns that start with a specified string. This makes it easy to remove all columns from one property, for example.

```
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin


class DropColumnsTransformer(BaseEstimator, TransformerMixin):
  _columns_to_delete: list[str]
  _starts_with: list[str] | str | None

  def __init__(self, columns_to_delete: list | None = None, starts_with: list[str]
    self._columns_to_delete = columns_to_delete if columns_to_delete is not None els
    self._starts_with = []

    if starts_with is not None:
      if type(starts_with) == str:
        self._starts_with = [starts_with]
      if type(starts_with) == list:
        self._starts_with = starts_with

  def fit(self, X: pd.DataFrame, y=None):
    return self

  def _get_columns_to_delete(self, X: pd.DataFrame):
    columns_to_delete = []
    for col in X.columns:
      if col in self._columns_to_delete:
        columns_to_delete.append(col)
        continue
      if any([col.startswith(starts_with) for starts_with in self._starts_with]):
        columns_to_delete.append(col)
        continue

    return columns_to_delete

  def transform(self, X: pd.DataFrame):
    columns_to_delete = self._get_columns_to_delete(X)
    return X.drop(columns=columns_to_delete)
```

FillPropertyNaNsTransformer

The `FillPropertyNaNsTransformer` fills NaN values in a DataFrame based on the specified parameter and fill strategy. The transformer supports the following fill strategies:

- `mean` - Fill NaN values with the mean of the column
- `median` - Fill NaN values with the median of the column
- `zero` - Fill NaN values with zero
- `interpolate` - Fill NaN values by interpolating between the nearest values with optional limit
- `ffill` - Fill NaN values by forward filling with optional limit
- `bfill` - Fill NaN values by backward filling with optional limit

The methods can be passed as a list to apply multiple fill strategies in sequence.

```python
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin
from typing import Literal

parameters = ['bg', 'insulin', 'carbs', 'hr', 'steps', 'cals', 'activity']
time_diffs = [f'-{i}:{j:02}' for i in range(6) for j in range(0, 60, 5)]

Parameter = Literal['bg', 'insulin', 'carbs', 'hr', 'steps', 'cals', 'activity']
How = Literal['mean', 'median', 'zero', 'interpolate']


class FillPropertyNaNsTransformer(BaseEstimator, TransformerMixin):
  _parameter: Parameter | str
  _hows: list[How] | list[str]
  _mean: None | float
  _median: None | float
  _precision: None | int
  _ffill: bool | int
  _bfill: bool | int
  _interpolate: bool | int

  def __init__(self,
               parameter: Parameter | str,
               how: How | list[How] | str | list[str] = 'median',
               precision: int | None = None,
               ffill: bool | int = True,
               bfill: bool | int = True,
               interpolate: bool | int = True):
    if not parameter in parameters:
      raise ValueError(f'parameter must be one of {parameters}')

    hows = how if isinstance(how, list) else [how]
    for how in hows:
      if not how in ['mean', 'median', 'zero', 'interpolate']:
        raise ValueError(f'how must be one of mean, median, zero, interpolate')

    self._parameter = parameter
    self._hows = hows
    self._precision = precision
    self._ffill = ffill
    self._bfill = bfill
    self._interpolate = interpolate

  def fit(self, X: pd.DataFrame, y=None):
    self._mean = X[self._get_affection_columns(X=X)].stack().mean(numeric_only=True
    self._median = X[self._get_affection_columns(X=X)].stack().median(numeric_only=
    return self

  def _get_affection_columns(self, X: pd.DataFrame):
    columns = X.columns
    affected_columns = []
    for time_diff in time_diffs:
      affected_column = f'{self._parameter}{time_diff}'
```

```
      if affected_column in columns:
        affected_columns.append(affected_column)

    return affected_columns

  def transform(self, X: pd.DataFrame):
    X = X.copy()
    columns = self._get_affection_columns(X)
    for how in self._hows:
      if how == 'mean':
        X[columns] = X[columns].fillna(self._mean)

      if how == 'median':
        X[columns] = X[columns].fillna(self._median)

      if how == 'zero':
        X[columns] = X[columns].fillna(0)

      if how == 'interpolate':
        limit = self._interpolate if type(self._interpolate) == int else None
        X[columns] = X[columns].interpolate(axis=1, limit=limit)

        if self._ffill:
          limit = self._ffill if type(self._ffill) == int else None
          X[columns] = X[columns].ffill(axis=1, limit=limit)

        if self._bfill:
          limit = self._bfill if type(self._bfill) == int else None
          X[columns] = X[columns].bfill(axis=1, limit=limit)

        if self._precision:
          X[columns] = X[columns].round(self._precision)

    return X
```

PropertyOutlierTransformer

The `PropertyOutlierTransformer` removes outliers from a DataFrame based on a specified parameter and filter function. The transformer supports the following fill strategies:

- `min` - Fill NaN values with the minimum of the column
- `max` - Fill NaN values with the maximum of the column
- `mean` - Fill NaN values with the mean of the column
- `median` - Fill NaN values with the median of the column
- `zero` - Fill NaN values with 0

```python
import pandas as pd
from sklearn.base import BaseEstimator, TransformerMixin
from typing import Literal, Callable

parameters = ['bg', 'insulin', 'carbs', 'hr', 'steps', 'cals', 'activity']

Parameter = Literal['bg', 'insulin', 'carbs', 'hr', 'steps', 'cals', 'activity']
FillStrategy = Literal['min', 'max', 'mean', 'median', 'zero']


class PropertyOutlierTransformer(BaseEstimator, TransformerMixin):
  _parameter: Parameter
  _fill_strategy: FillStrategy
  _filter_function: Callable

  def __init__(self, parameter: Parameter, filter_function: Callable, fill_strategy
    if not parameter in parameters:
      raise ValueError(f'parameter must be one of {parameters}')

    self._parameter = parameter
    self._filter_function = filter_function
    self._fill_strategy = fill_strategy

  def fit(self, X: pd.DataFrame, y=None):
    return self

  def _get_affection_columns(self, X: pd.DataFrame):
    affected_columns = [col for col in X.columns if self._parameter in col]
    return affected_columns

  def _fill_outlier(self, x, X: pd.DataFrame, column: str):
    if self._fill_strategy == 'zero':
      return 0
    if self._fill_strategy == 'min':
      return x if x is not None else X[column].min()
    if self._fill_strategy == 'max':
      return x if x is not None else X[column].max()
    if self._fill_strategy == 'mean':
      return x if x is not None else X[column].mean()
    if self._fill_strategy == 'median':
      return x if x is not None else X[column].median()

  def transform(self, X: pd.DataFrame):
    X = X.copy()
    columns = self._get_affection_columns(X)
    for column in columns:
      X[column] = X[column].apply(lambda x: self._fill_outlier(x, X, column) if sel

    return X
```

Model HyperParameter tunning

To ensure consistent and efficient tuning of different models tested, we created a series of **HyperParameter Tuner** classes for the best-performing models identified through LazyPredict. These different classes automatically perform Grid Search using BayesSearchCV and apply a **5-fold cross-validation** strategy to identify an optimal set of hyperparameters.

Example of Tuner Class: XGBoost

```python
from xgboost import XGBRegressor
from skopt.space import Integer, Real

from src.features.tuners.BaseHyperparameterTuner import BaseHyperparameterTuner

param_spaces = {
    'default': {
        'n_estimators': Integer(100, 300),  # Common range for boosting rounds
        'learning_rate': Real(0.03, 0.1, 'log-uniform'),  # Standard learning rates
        'max_depth': Integer(3, 7),  # Default depth range that balances depth and
        'min_child_weight': Integer(1, 5),  # Moderate range to allow small variati
        'subsample': Real(0.7, 0.85),  # Light subsampling for diversity while pres
        'colsample_bytree': Real(0.7, 0.85),  # Common column sampling range
        'gamma': Integer(0, 5),  # Small gamma range for light regularization
        'alpha': Real(1e-3, 0.1, 'log-uniform'),  # Default L1 regularization
        'lambda': Real(1e-3, 0.1, 'log-uniform'),  # Default L2 regularization
    },
    'deep': {
        'n_estimators': Integer(50, 1000),  # Number of boosting rounds
        'learning_rate': Real(0.001, 0.3, 'log-uniform'),  # Step size shrinkage, l
        'max_depth': Integer(3, 15),  # Maximum tree depth for base learners
        'min_child_weight': Integer(1, 10),  # Minimum sum of instance weight (hess
        'subsample': Real(0.5, 1.0),  # Subsample ratio of the training instances
        'colsample_bytree': Real(0.3, 1.0),  # Subsample ratio of columns when cons
        'gamma': Integer(0, 10),  # Minimum loss reduction required to make a furth
        'alpha': Real(1e-10, 10.0, 'log-uniform'),  # L1 regularization term on wei
        'lambda': Real(1e-10, 10.0, 'log-uniform'),  # L2 regularization term on we
        'scale_pos_weight': Integer(1, 100),  # Controls balance of positive and ne
    },
    'fast': {
        'n_estimators': [100, 200],  # Limited boosting rounds for quick training
        'learning_rate': [0.05, 0.1],  # Common default learning rates
        'max_depth': [3, 5],  # Shallow trees to keep training fast and avoid overf
        'min_child_weight': [1, 3],  # Minimal regularization range
        'subsample': [0.8],  # Fixed sampling value; often effective
        'colsample_bytree': [0.8],  # Fixed column sampling
        'gamma': [0, 1],  # Light regularization for basic pruning
    },
    'custom': {
        'n_estimators': Integer(1500, 5000),  # Allow for more boosting rounds
        'learning_rate': Real(0.01, 0.05, 'log-uniform'),  # Slightly higher learni
        'max_depth': Integer(5, 7),  # Allow deeper trees
        'min_child_weight': Integer(2, 7),  # Explore smaller values
        'subsample': Real(0.5, 0.8),  # Light subsampling for diversity while prese
        'colsample_bytree': Real(0.6, 0.9),  # Common column sampling range
        'gamma': Real(0.1, 2, 'log-uniform'),  # Small gamma range for light regula
        'alpha': Real(1, 50, 'log-uniform'),  # Increase upper bound for L1 regular
        'lambda': Real(1, 20, 'log-uniform'),  # Default L2 regularization
    }
}


class XGBHyperparameterTuner(BaseHyperparameterTuner):
```

```
    __name__ = 'XGBRegressor'

    @staticmethod
    def regressor() -> XGBRegressor:
        return XGBRegressor(objective='reg:squarederror', random_state=42, tree_metl

    @staticmethod
    def param_space(search_space: str | None) -> dict | None:
        if search_space is None:
            return None
        return param_spaces[search_space] if search_space in param_spaces.keys() els

    def fit(self, X_train, y_train, X_test=None, y_test=None):
        super().fit(X_train, y_train, X_test, y_test)
```

Custom Splitter

The custom splitter is a class that extends the `BaseCrossValidator` class from `scikit-learn`. It is used to split the data into training and test sets based on the group values provided during fitting. The custom splitter is used in the `BayesSearchCV` class to perform cross-validation with group-based splits.

Groups set to `0` are always included in the training set, while groups set to `1` are split into training and test sets based on the specified test size. The custom splitter uses the `ShuffleSplit` class from scikit-learn to randomly split the group 1 data into training and test sets.

With the help of the custom splitter, we can ensure that the train data is always included in the training set, while the augmented data is used with 80% in the training set and 20% in the test set. This allows us to evaluate the model's performance on the augmented data and ensure that the model generalizes well to unseen data.

Source Code

```
import numpy as np
from sklearn.model_selection import BaseCrossValidator, ShuffleSplit


class CustomSplitter(BaseCrossValidator):
  def __init__(self, test_size=0.2, n_splits=1, random_state=None):
    """
    Custom splitter for BayesSearchCV.

    Parameters:
    - test_size (float): Proportion of the data with group 1 to use in the test set
    - n_splits (int): Number of splits to generate.
    - random_state (int or None): Random seed for reproducibility.
    """
    self.test_size = test_size
    self.n_splits = n_splits
    self.random_state = random_state
    self._groups = None

  def fit(self, X, y=None, groups=None):
    """
    Store the group list provided during fitting.
    Parameters:
    - X: Input data (not used, included for API compatibility).
    - y: Target values (not used, included for API compatibility).
    - groups (array-like): List of group values (0, 1).
    """
    if groups is None:
      raise ValueError("Groups must be provided.")
    self._groups = np.asarray(groups)
    return self

  def split(self, X, y=None, groups=None):
    """
    Generate train/test splits.

    Parameters:
    - X: Input data.
    - y: Target values (not used).
    - groups (ignored, the stored groups from `fit` are used).

    Yields:
    - train_indices: Indices for the training set.
    - test_indices: Indices for the test set.
    """
    if self._groups is None:
      self.fit(X, groups=groups)

    # Indices for groups
    _group_0_indices = np.where(self._groups == 0)[0]
    _group_1_indices = np.where(self._groups == 1)[0]

    if len(_group_1_indices) == 0:
```

```
            raise ValueError("Group 1 has no samples to split. Please ensure that group 1

        # Use ShuffleSplit for random splitting of group 1
        shuffle_split = ShuffleSplit(n_splits=self.n_splits, test_size=self.test_size,

        for train_1_indices, test_1_indices in shuffle_split.split(_group_1_indices):
            # Map the shuffled indices back to the original group_1_indices
            _train_1 = _group_1_indices[train_1_indices]
            _test_1 = _group_1_indices[test_1_indices]

            # Combine train indices
            train_indices = np.concatenate([_group_0_indices, _train_1])
            yield train_indices, _test_1

    def get_n_splits(self, X=None, y=None, groups=None):
        """
        Returns the number of splits.
        """
        return self.n_splits
```

Bibliography and References

[1] Oct 2024. URL: https://en.wikipedia.org/wiki/Diabetes.

[2] Conor Seery. Normal and diabetic blood sugar level ranges. Nov 2022. URL: https://www.diabetes.co.uk/diabetes_care/blood-sugar-level-ranges.html.

[3] Adam Brown. 42 factors that affect blood glucose?! a surprising update. Sep 2022. URL: https://diatribe.org/diabetes-management/42-factors-affect-blood-glucose-surprising-update.

[4] Hoda Nemat, Heydar Khadem, Jackie Elliott, and Mohammed Benaissa. Data-driven blood glucose level prediction in type 1 diabetes: a comprehensive comparative analysis. *Scientific Reports*, 14(1):21863, Jan 2024. URL: https://doi.org/10.1038/s41598-024-70277-x, doi:10.1038/s41598-024-70277-x.

[5] Cindy Marling and Razvan Bunescu. The OhioT1DM dataset for blood glucose level prediction: update 2020. *CEUR Workshop Proc.*, 2675:71–74, September 2020.