# 2024_TJU_Data_Mining-Analysis_Report

# Blood Glucose Prediction Model Based on Time Series Convolutional Neural Network

# 1 Issue Analysis

## 1.1 Brief Introduction

This paper aims to construct a blood glucose level time series prediction model to assist diabetes patients in managing their blood sugar levels. Firstly, we delve into understanding the mechanisms of blood glucose regulation, factors influencing blood glucose levels, and the management methods for different types of diabetes. Secondly, based on the relevant data from the `Shanghai_T1DM` and `Shanghai_T2DM` datasets, we perform data cleaning, handle missing and outlier values, and conduct necessary data transformations and normalization. In terms of feature engineering, we select or construct features critical to glucose prediction based on domain knowledge, explore correlations among features, and perform feature selection or dimensionality reduction to enhance model efficiency. Next, we investigate and compare machine learning models suitable for time series glucose prediction, with the flexibility to choose methods from the latest papers or classical and effective approaches. The goal is to implement the selected model and fine-tune parameters to optimize prediction performance. Subsequently, we evaluate the model's predictive accuracy and generalization capacity using techniques such as cross-validation, AUC-ROC curves, and mean squared error. Finally, we conduct a thorough analysis of prediction outcomes, examining the impact of different factors on prediction accuracy, and present prediction results and model performance clearly using graphs and visualization tools. The aim is to predict blood glucose levels at 15, 30, 45, and 60 minutes and document the analysis results in this report.

## 1.2 Current Mainstream Blood Glucose Prediction Methods

### 1.2.1 Nonlinear Autoregressive (NAR) Neural Network

According to research by Alessandro Aliberti and his team (source), they developed an efficient blood glucose prediction method based on the Nonlinear Autoregressive (NAR) neural network. Unlike traditional linear autoregressive models, the `NAR` model is not constrained by distribution and can handle nonlinear features in blood glucose data, such as sudden fluctuations and transient periods. This method uses past blood glucose values as input and employs a multi-layer neural network to compute future glucose levels. The initial design includes an input layer, a hidden layer, and an output layer, trained using the `Levenberg-Marquardt` backpropagation process. To optimize the model, they employed the `Lipschitz` method to determine the optimal number of regressors and used the Optimal Brain Surgeon method for automatic pruning to enhance model compactness and generalization. After experimental validation, they selected a final `NAR` model with 8 regressors to achieve precise and efficient blood glucose level predictions.

### 1.2.2 Long Short-Term Memory (LSTM) and Recurrent Neural Network (RNN)

According to research by Mario Munoz-Organero and his team (source), they predicted changes in blood glucose levels using Long Short-Term Memory (LSTM) and Recurrent Neural Network (RNN) models. The model integrates current blood glucose levels, carbohydrate intake, and insulin injections, describing the glucose metabolism process using differential equations including digestion, absorption, insulin-dependent and independent utilization, renal clearance, and endogenous liver production. By training the `RNN` to learn carbohydrate digestion and insulin absorption processes, and the `LSTM` model to learn time patterns of blood glucose levels, combined with processed insulin and carbohydrate signals, they estimated future continuous glucose monitoring (CGM) readings' glucose changes. The model generates predictions based on metabolic dynamics without external inputs, providing early warnings of potential adverse situations and recommending specific measures to avoid them.

#### 1.2.3 Autoregressive Moving Average (ARMA) Model

According to research by Ning Ma and his team (source), their ARMA model combines autoregressive (AR) and moving average (MA) components to capture linear and nonlinear dynamics in time series data. The model first updates input data using sliding window techniques and determines the optimal model order using `AIC` and `BIC`. Subsequently, comparing predicted results with original data yields residual time series. The residuals are then predicted using a BP neural network to enhance the model's adaptation to nonlinear components. Finally, combining `ARMA` model predictions with residual compensation network results through statistical analysis yields the final blood glucose predictions. This framework effectively handles linear and nonlinear features in blood glucose data, enhancing prediction accuracy and stability suitable for real-time blood glucose control and management.

### 1.3 Team Member

2151617 Zheng Zhi 2152970 Li Jinlin 2154306 Li Zekai 2154314 Zheng Kai

### 1.4 Code Management

We use `Github` for code management and version control in this project. All source code, environments, report documents, and PPT slides are stored in the following repository:

github.com/MouzKarrigan/2024_TJU_Data_Mining-Analysis

## 2 Dataset Analysis

### 2.1 Dataset Introduction

The datasets `ShanghaiT1DM` and `ShanghaiT2DM` comprise two folders named `Shanghai_T1DM` and `Shanghai_T2DM` and two summary sheets named `Shanghai_T1DM_Summary.csv` and `Shanghai_T2DM_Summary.csv`.

The `Shanghai_T1DM` folder and `Shanghai_T2DM` folder contain 3 to 14 days of `CGM` data corresponding to 12 patients with T1DM and 100 patients with T2DM, respectively. Of note, for one patient, there might be multiple periods of `CGM` recordings due to different visits to the hospital, which were stored in different excel tables. In fact, collecting data from different periods in one patient can reflect the changes of diabetes status during the follow-up. The excel table is named by the `patient ID`, `period number` and the `start date` of the `CGM` recording. Thus, for 12 patients with T1DM, there are 8 patients with 1 period of the `CGM` recording

and 2 patients with 3 periods, totally equal to 16 excel tables in the `Shanghai_T1DM` folder. As for 100 patients with T2DM, there are 94 patients with 1 period of `CGM` recording, 6 patients with 2 periods, and 1 patient with 3 periods, amounting to 109 excel tables in the `Shanghai_T2DM` folder. Overall, the excel tables include `CGM` BG values every 15 minutes, capillary blood glucose `CBG` values, `blood ketone`, self-reported `dietary intake`, `insulin doses` and `non-insulin hypoglycemic agents`. The `blood ketone` was measured when diabetic ketoacidosis was suspected with a considerably high glucose level. Insulin administration includes continuous subcutaneous insulin infusion using insulin pump, multiple daily injections with insulin pen, and insulin that were given intravenously in case of an extremely high BG level.

Each excel table in the `Shanghai_T1DM` folder and `Shanghai_T2DM` folder contains the following data fields: `Date` Recording time of the `CGM` data. `CGM` CGM data recorded every 15 minutes. `CBG` CBG level measured by the glucose meter. `Blood ketone` Plasma-hydroxybutyrate measured with ketone test strips (Abbott Laboratories, Abbott Park, Illinois, USA). `Dietary intake` Self-reported time and weighed food intake `Insulin dose-s.c.` Subcutaneous insulin injection with insulin pen. `Insulin dose-i.v.` Dose of intravenous insulin infusion. `Non-insulin hypoglycemic agents` Hypoglycemic agents other than insulin. `CSII-bolus insulin` Dose of insulin delivered before a meal through insulin pump. `CSII-basal insulin` The rate (iu/per hour) at which basal insulin was continuously infused through insulin pump.

The summary sheets summarize the clinical characteristics, laboratory measurements and medications of the patients included in this study, with each row corresponding to one excel table in `Shanghai_T1DM` and `Shanghai_T2DM` folders. Clinical characteristics include `patient ID`, `gender`, `age`, `height`, `weight`, `BMI`, `smoking and drinking history`, `type of diabetes`, `duration of diabetes`, `diabetic complications`, `comorbidities` as well as `occurrence of hypoglycemia`. Laboratory measurements contain fasting and 2-hour postprandial plasma glucose/C-peptide/insulin, hemoglobin A1c (HbA1c), glycated albumin, total cholesterol, triglyceride, high-density lipoprotein cholesterol, low-density lipoprotein cholesterol, creatinine, estimated glomerular filtration rate, uric acid and blood urea nitrogen. Both hypoglycemic agents and medications given for other diseases before the `CGM` reading were also recorded.

## 2.2 Data Pre-process

### 2.2.1 Format Transformation

For ease of subsequent pre-processing, first convert all .xlsx files in the T1DM and T2DM datasets to .csv files and store them.

```
for file in T1DM:
    if file == '.DS_Store':
        continue
    data = pd.read_excel(os.path.join(T1DM_folder_url, file))

    new_filename = file.split('.')[0] + '.csv'
    data.to_csv(os.path.join(new_T1DM_folder_url, new_filename))
```

### 2.2.2 Dietary Intake Pre-process

In the original dataset, there are two attributes representing the patient's `Dietary Intake` content and its Chinese translation. We will combine these two attributes into one, simply determining whether there is

content in the original `Dietary Intake` data. If there is content, it indicates that the patient is eating at that time, and we mark it as `1` in the output data's `Dietary Intake`. If there is no content, we mark it as `0` in the output data.

```
def dietary_intake_process(df: DataFrame) -> DataFrame:
    attribute_name = 'Dietary intake'
    df_attribute = df[attribute_name]
    new_df_attribute = df_attribute.copy()
    for index, row in enumerate(df_attribute):
        if pd.isna(row):
            new_df_attribute[index] = 0
        else:
            new_df_attribute[index] = 1
    df[attribute_name] = new_df_attribute
    # print(new_df_attribute)
    return df
```

We ignore the specific content of the meals and only consider whether the patient is eating or not, aiming to simplify the subsequent model construction and computation. Given the limited data, analyzing the average impact of eating on blood glucose levels is more accurate and convenient than analyzing the varying effects of each specific nutrient intake on blood glucose levels.

### 2.2.3 Agent Details Pre-process

In the original dataset, `Shanghai_T1DM_Summary.csv` and `Shanghai_T2DM_Summary.csv` provide information on all the medications used in the dataset and whether they contain insulin. To facilitate subsequent data preprocessing, we identified and extracted all medication names using `agents_process.py` and saved them in `agents_info.json`.

```
urls = ['Shanghai_T1DM_Summary.csv', 'Shanghai_T2DM_Summary.csv']
agents = set()
agents_list = []

for url in urls:
    full_url = os.path.join('raw-data', url)
    df = pd.read_csv(full_url)
    df_agents = df["Hypoglycemic Agents"]
    for i in df_agents:
        some_agents = i.split(',')
        for j in some_agents:
            if j != 'none':
                agents.add(j.strip())


for agent in agents:
    agents_list.append(agent)

agents_list_sorted = sorted(agents_list, key=len)
```

```
    agent_str = json.dumps(agents_list, indent=2)

    with open('agents_info.json', 'w') as file:
        file.write(agent_str)
```

Then we extracted and saved the names of insulin-containing antidiabetic medications to `insulin_agents.json`, and the names of non-insulin antidiabetic medications to `non_insulin_agents.json`.

### 2.2.4 Insulin Dose - s.c. Pre-process

In the original dataset, there is an attribute `Insulin Dose - s.c.` that records the medication name and dosage administered via subcutaneous injection in the format `Novolin R, 2 IU`, with the medication name before the comma and the dosage after the comma. To handle this, we designed a method to separately extract the medication name and dosage from the `Insulin Dose - s.c.` attribute and record them in the output data.

```
def insulin_dose_sc_process(df: DataFrame) -> DataFrame:
    attribute_name = 'Insulin dose - s.c.'
    df_insulin_attribute = []
    with open('insulin_agents.json', 'r') as file:
        agents_map = json.loads(file.read())
    for a in agents_map:
        df_insulin_attribute.append(attribute_name + ' ' + a)
    df_insulin_dose = pd.DataFrame(columns=df_insulin_attribute,
index=range(df.shape[0]))
    df_attribute = df[attribute_name]
    for index, row in enumerate(df_attribute):
        for a in df_insulin_attribute:
                df_insulin_dose[a][index] = 0
        if not pd.isna(row):
            agents = row.split(';')
            for _, agent in enumerate(agents):
                for a in agents_map:
                    if re.search(a, agent):
                        ds = re.sub(a, '', agent)
                        ds = re.sub(',', '', ds)
                        ds = re.sub('IU', '', ds)
                        ds = ds.strip()
                        df_insulin_dose[attribute_name + ' ' + a][index] = ds
                        pattren = r'\d+'
                        if not re.match(pattren, ds):
                            print(ds)


    df = df.drop(columns=[attribute_name])
    df = pd.concat([df, df_insulin_dose], axis=1)
    return df
```

It is worth noting that a single injection may not always involve only one type of medication but could simultaneously include two or more different types. Therefore, in the preprocessed output data, we do not use the conventional method of converting the `Insulin Dose - s.c.` attribute into `Insulin Kind` and `Dose` attributes. Instead, we set all the insulin-containing antidiabetic medication names that appear in `insulin_agents.json` as new attributes. In the output data, we only record the dosage under the corresponding medication type attribute, with the unit in IU. If a particular medication is not present, the dosage is recorded as `0`. This approach allows us to handle cases where two or more different types of medications are injected simultaneously.

### 2.2.5 Insulin Dose - i.v. Pre-process

In the original dataset, the attribute `Insulin Dose - s.c.` records the names and doses of medications administered to patients via intravenous injection, similar to `500ml 0.9% sodium chloride, 12 IU Novolin R, 10 ml 10% potassium chloride`. We disregard the irrelevant auxiliary medication components before and after, extracting only the main components of antidiabetic medications that affect blood glucose concentration from the middle of each record.

```
def insulin_dose_iv_process(df: DataFrame) -> DataFrame:
    attribute_name = 'Insulin dose - i.v.'
    df_insulin_attribute = []
    with open('insulin_agents.json', 'r') as file:
        agents_map = json.loads(file.read())
    for a in agents_map:
        df_insulin_attribute.append(attribute_name + ' ' + a)
    df_insulin_dose = pd.DataFrame(columns=df_insulin_attribute,
index=range(df.shape[0]))
    df_attribute = df[attribute_name]
    for index, row in enumerate(df_attribute):
        for a in df_insulin_attribute:
            df_insulin_dose[a][index] = 0
        if not pd.isna(row):
            agents = row.split(',')
            for _, agent in enumerate(agents):
                for a in agents_map:
                    if re.search(a, agent):
                        ds = re.sub(a, '', agent)
                        ds = re.sub('IU', '', ds)
                        ds = ds.strip()
                        df_insulin_dose[attribute_name + ' ' + a][index] = ds
                        pattren = r'\d+'
                        if not re.match(pattren, ds):
                            print(ds)


    df = df.drop(columns=[attribute_name])
    df = pd.concat([df, df_insulin_dose], axis=1)
    return df
```

The processing procedure is similar to that of 2.2.4. All antidiabetic medication names appearing in `insulin_agents.json` are set as new attributes separately. In the output data, only the doses are recorded under the corresponding medication type attribute, with the unit being IU. If there is no such medication, the dose is recorded as `0`. It's worth noting that, to differentiate between intravenous and subcutaneous injections, the newly added medication dose attributes will have prefixes `Insulin dose - s.c.` or `Insulin dose - i.v.` based on the injection form, facilitating subsequent model calculations.

### 2.2.6 Non-insulin Hypoglycemic Agents Pre-process

In the original dataset, the attribute `Non-insulin hypoglycemic agents` records the types and doses of non-insulin hypoglycemic agents consumed by patients. Our preprocessing of this attribute is similar to that of sections 2.2.4 and 2.2.5.

```
def non_insulin_hypoglycemic_process(df: DataFrame) -> DataFrame:
    attribute_name = 'Non-insulin hypoglycemic agents'
    df_non_insulin_attribute = []
    with open('non_insulin_agents.json', 'r') as file:
        agents_map = json.loads(file.read())
    for a in agents_map:
        df_non_insulin_attribute.append(attribute_name + ' ' + a)
    df_insulin_dose = pd.DataFrame(columns=df_non_insulin_attribute,
index=range(df.shape[0]))
    df_attribute = df[attribute_name]


    for index, row in enumerate(df_attribute):
        if not pd.isna(row):
            agents = row.split(' ')
            non_insulins = ''
            get_insulins = False

            for i, agent in enumerate(agents):
                if i % 3 == 0:
                    for a in agents_map:
                        if re.search(a, agent):
                            non_insulins = a
                            get_insulins = True
                elif i % 3 == 1:
                    if get_insulins == True:
                        dose = str(agent.strip())
                        try:
                            df_insulin_dose[attribute_name + ' ' + non_insulins]
[index] = dose
                        except UnboundLocalError as e:
                            print(attribute_name + ' ' + insulins)
                            exit(-1)

                        pattren = r'\d+'
                        if not re.match(pattren, dose):
                            print(dose)
```

```
                else:
                    insulins = ''
                    continue


    df = df.drop(columns=[attribute_name])
    df = pd.concat([df, df_insulin_dose], axis=1)
    return df
```

All newly added attributes in the output data are formed by prefixing the drug name with `Non-insulin hypoglycemic agents`, distinguishing them from the new attributes added in the processes of sections 2.2.4 and 2.2.5. These attributes only record the dosage of the respective drug.

### 2.2.7 CSII Pre-process

In the original dataset, the attribute `CSII - bolus insulin (Novolin R, IU)` and `CSII - basal insulin (Novolin R, IU / H)` exist. The latter represents the continuous basal insulin dose within a period, presented in merged cells in the .xlsx file. When converted to .csv format, some data loss occurred. We devised a method to address this issue by completing the missing data. Similarly, in the output data, the entire period's `CSII - basal insulin (Novolin R, IU / H)` is recorded, but each entry is recorded separately instead of being merged.

```
def basal_insulin_process(df: DataFrame) -> DataFrame:
    attribute_name = 'CSII - basal insulin (Novolin R, IU / H)'
    df_attribute = df[attribute_name]
    # print(df[attribute_name])
    new_df_attribute = df_attribute.copy()
    last_data = None
    for index, row in enumerate(df_attribute):
        if pd.isna(row):
            new_df_attribute[index] = last_data
        else:
            if row == 'temporarily suspend insulin delivery':
                last_data = 0
                new_df_attribute[index] = last_data
            else:
                last_data = row
    df[attribute_name] = new_df_attribute
    return df
```

It's worth noting that in both `CSII - bolus insulin (Novolin R, IU)` and `CSII - basal insulin (Novolin R, IU / H)` attributes, there might be records indicating "temporarily suspend insulin delivery," signifying a pause in continuous medication delivery for the next short period. We devised a method to identify and record the subsequent period's dose as `0` in the output data until new dosage information is available.

```
def bolus_insulin_process(df: DataFrame) -> DataFrame:
    attribute_name = 'CSII - bolus insulin (Novolin R, IU)'
```

```
    df_attribute = df[attribute_name]
    # print(df[attribute_name])
    new_df_attribute = df_attribute.copy()
    for index, row in enumerate(df_attribute):
        if pd.isna(row):
            new_df_attribute[index] = 0
        else:
            if row == 'temporarily suspend insulin delivery':
                new_df_attribute[index] = 0
    df[attribute_name] = new_df_attribute
    return df
```

### 2.2.8 Other Attributes Pre-process

In the original dataset, other attributes such as `Date` and `CGM (mg/dl)` are directly usable, so we transferred them to the output data.

```
def read_csv(url) -> DataFrame:
    return pd.read_csv(url)

def save_csv(df: DataFrame, url: str):
    df.to_csv(url)
```

But the attribute `CBG (mg/dl)` refers to blood glucose concentration measured in another way, with many empty values that are difficult to handle in the model. Also, it overlaps with `CGM (mg/dl)`, so we removed the `CBG (mg/dl)` attribute from the output data. `Blood Ketone (mmol/L)` indicates blood ketone levels, which have poor relevance to the problem at hand and many missing values, making it challenging to handle. Therefore, we also removed this attribute from the output data.

```
def process_CBG_blood_ketone(df:DataFrame) -> DataFrame:
    return df.drop(columns=['CBG (mg / dl)', 'Blood Ketone (mmol / L)'])
```

### 2.2.9 Data Select

In several .xlsx files in the original dataset, some tables contain serious formatting errors due to oversight by the researchers during recording. Considering the minimal impact on a very small portion of the data and the associated development costs, we opted not to devise additional methods to handle data with severe formatting issues but to exclude them directly. The specific actions and reasons are as follows.

```
ban_url = [
    '2045_0_20201216.csv',  # CSII - bolus insulin without unit
    '2095_0_20201116.csv',  # CSII - bolus insulin without unit
    '2013_0_20220123.csv',  # No dietary intake, only meal intake
    '2027_0_20210521.csv',  # Basal insulin dose in Chinese

]
```

### 2.2.10 Pre-Processed Data

Using the various methods mentioned above, we preprocessed all the data in the `Shanghai_T1DM` and `Shanghai_T2DM` datasets, and the results are saved in the `processed-data` folder. The attributes of the processed data are as follows.

```
Date,CGM (mg / dl),Dietary intake,"CSII - bolus insulin (Novolin R, IU)","CSII -
basal insulin (Novolin R, IU / H)",Insulin dose - s.c. insulin aspart
70/30,Insulin dose - s.c. insulin glarigine,Insulin dose - s.c. Gansulin R,Insulin
dose - s.c. insulin aspart,Insulin dose - s.c. insulin aspart 50/50,Insulin dose -
s.c. Humulin 70/30,Insulin dose - s.c. insulin glulisine,Insulin dose - s.c.
Novolin 30R,Insulin dose - s.c. Novolin 50R,Insulin dose - s.c. insulin
glargine,Insulin dose - s.c. Humulin R,Insulin dose - s.c. insulin
degludec,Insulin dose - s.c. Gansulin 40R,Insulin dose - s.c. Novolin R,Insulin
dose - s.c. insulin detemir,Non-insulin hypoglycemic agents acarbose,Non-insulin
hypoglycemic agents gliquidone,Non-insulin hypoglycemic agents sitagliptin,Non-
insulin hypoglycemic agents voglibose,Non-insulin hypoglycemic agents
repaglinide,Non-insulin hypoglycemic agents liraglutide,Non-insulin hypoglycemic
agents glimepiride,Non-insulin hypoglycemic agents pioglitazone,Non-insulin
hypoglycemic agents canagliflozin,Non-insulin hypoglycemic agents
dapagliflozin,Non-insulin hypoglycemic agents gliclazide,Non-insulin hypoglycemic
agents metformin,Insulin dose - i.v. insulin aspart 70/30,Insulin dose - i.v.
insulin glarigine,Insulin dose - i.v. Gansulin R,Insulin dose - i.v. insulin
aspart,Insulin dose - i.v. insulin aspart 50/50,Insulin dose - i.v. Humulin
70/30,Insulin dose - i.v. insulin glulisine,Insulin dose - i.v. Novolin
30R,Insulin dose - i.v. Novolin 50R,Insulin dose - i.v. insulin glargine,Insulin
dose - i.v. Humulin R,Insulin dose - i.v. insulin degludec,Insulin dose - i.v.
Gansulin 40R,Insulin dose - i.v. Novolin R,Insulin dose - i.v. insulin detemir
```

# 3 Blood Glucose Prediction Model

## 3.1 Model Construction

To predict the blood glucose levels of any patient at 15, 30, 45, and 60 minutes based on the time series and static data provided in the `ShanghaiT1DM` and `ShanghaiT2DM` datasets, we chose the `LSTM` layer for processing all time series data due to its outstanding performance in capturing dependencies and features in sequential data through convolution and residual connections. This includes `drug concentrations at any given time`, `medication information`, `dietary intake`, and the patient's previous `CGM` data. Additionally, the `Flatten` layer ensures that the embedding vectors of the time series data are flattened into structured N-dimensional vectors required for subsequent processing.

```
x = LSTM(64, return_sequences=True)(ts_input)
x = LSTM(56, return_sequences=True)(x)
x = LSTM(48, return_sequences=True)(x)
x = LSTM(40, return_sequences=True)(x)
x = LSTM(36, return_sequences=True)(x)
```

```
x = LSTM(32)(x)
ts_embedding = Flatten()(x)
```

All static data, including the patient's `ID`, `height`, `weight`, and `gender`, is processed through a simple `Dense` layer. This ensures that the static data's influence on blood glucose levels is considered by the prediction model and has the same embedding dimension N as the processed time series data.

```
y = Dense(64, activation='relu')(static_input)
y = Dense(56, activation='relu')(y)
y = Dense(48, activation='relu')(y)
y = Dense(40, activation='relu')(y)
y = Dense(36, activation='relu')(y)
y = Dense(32, activation='relu')(y)
y = Dense(32, activation='relu')(y)  # 保持 32 维
y = Dense(32, activation='relu')(y)  # 保持 32 维
static_embedding = y
```

Next, we designed a `cross_attention` layer that calculates the attention scores between the `ts_embedding` of the time series data and the `static_embedding` of the static data using matrix multiplication. The scores are then processed with `Softmax` to obtain weights, which are used to compute a weighted sum. After flattening, the output is a structured N-dimensional vector. In this way, the N-dimensional time series data and static data are encoded. By utilizing the `TCN`, `Dense`, and `cross_attention` layers, the model comprehensively considers the influence of all factors on blood glucose levels.

```
# Cross-Attention层
def cross_attention(query, key, value):
    attention_scores = tf.matmul(query, key, transpose_b=True)
    attention_weights = tf.nn.softmax(attention_scores, axis=-1)
    attended_vector = tf.matmul(attention_weights, value)
    return attended_vector

# 使用静态特征作为查询，时序特征作为键和值
query1 = tf.expand_dims(static_embedding, axis=1)
key1 = tf.expand_dims(ts_embedding, axis=1)
value1 = tf.expand_dims(ts_embedding, axis=1)
cross_attention_output1 = cross_attention(query1, key1, value1)
cross_attention_output1 = Flatten()(cross_attention_output1)

# 使用时序特征作为查询，静态特征作为键和值
query2 = tf.expand_dims(ts_embedding, axis=1)
key2 = tf.expand_dims(static_embedding, axis=1)
value2 = tf.expand_dims(static_embedding, axis=1)
cross_attention_output2 = cross_attention(query2, key2, value2)
cross_attention_output2 = Flatten()(cross_attention_output2)
```

In the decoding part, we further process the previously encoded N-dimensional vector using a `Dense` layer with an output dimension of 64 and a ReLU activation function. Then, we output four target values to predict

the blood glucose levels for the next 15, 30, 45, and 60 minutes.

```
# 合并两个 cross-attention 输出
merged_attention_output = Concatenate()([cross_attention_output1,
cross_attention_output2])

# 解码层，从合并后的维度逐渐减少到 4
z = Dense(64, activation='relu')(merged_attention_output)
z = Dense(56, activation='relu')(z)
z = Dense(48, activation='relu')(z)
z = Dense(40, activation='relu')(z)
z = Dense(36, activation='relu')(z)
z = Dense(32, activation='relu')(z)
z = Dense(16, activation='relu')(z)
output = Dense(4)(z)  # 输出层，预测4个目标值
```

Finally, we use the functions provided in the `TensorFlow` library to build, compile, and run the aforementioned model.

```
model = Model(inputs=[ts_input, static_input], outputs=output)
model.compile(optimizer='adam', loss='mean_squared_error', metrics=
['mean_absolute_error'])
model.summary()
```

During model execution, it will read the preprocessed data from previous steps, further process the data by separating the temporal features and prediction targets, and normalize the input data to facilitate subsequent training.

```
# Separating the temporal features and prediction targets
time_series_features = data[time_serise_attribute].values
static_features = data[static_attribute].values
targets = data[target_attribute].values

def create_sequences(features, targets, static_features, time_steps=10):
    ts_X,  y = [], []
    for i in range(len(features) - time_steps):
        ts_X.append(features[i:i+time_steps])
        static_X = static_features[i]
        y.append(targets[i+time_steps])
    return np.array(ts_X), np.array(static_X), np.array(y)

ts_X, static_X, y = create_sequences(time_series_features, targets,
static_features)

# Normalize the input data
scaler_ts_X = StandardScaler()
ts_X = scaler_ts_X.fit_transform(ts_X.reshape(-1,
ts_X.shape[-1])).reshape(ts_X.shape)
```

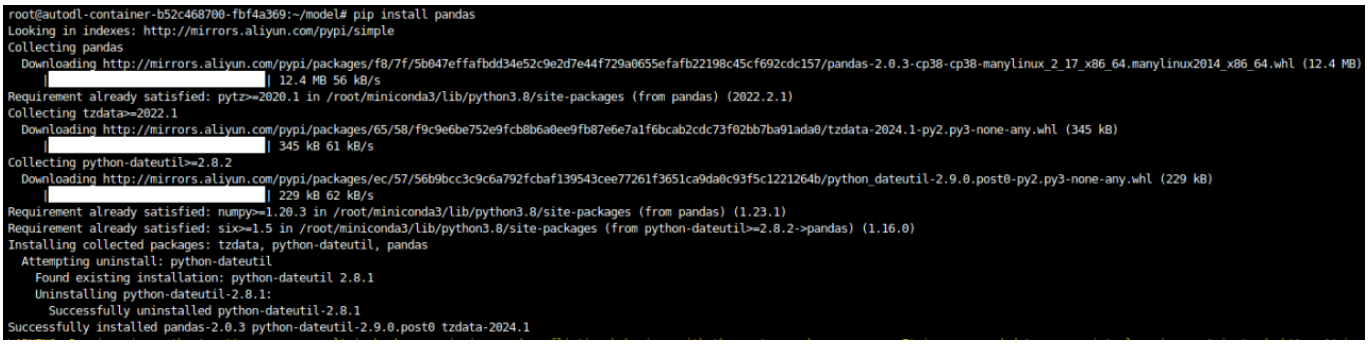
```
scaler_static_X = StandardScaler()
static_X = scaler_static_X.fit_transform(static_X)

scaler_y = StandardScaler()
y = scaler_y.fit_transform(y)

self.ts_X_train, self.ts_X_test, self.static_X_train, self.static_X_test,
self.y_train, self.y_test = train_test_split(
    ts_X, static_X, y, test_size=0.2, random_state=42
)
```

## 3.2 Model Deployment

Due to the computational limitations of the local machine and to save time, we chose to deploy the model on the cloud-based `AutoDL` platform. We rented an `RTX 4090D (24GB)` GPU, with a CPU configuration of `15 vCPU Intel(R) Xeon(R) Platinum 8474C`, on an hourly basis. The image version used was `TensorFlow 2.9.0 Python 3.8 (ubuntu 20.04) Cuda 11.2`. The specific configuration of the cloud model container is shown in the following image:


We then used `JupyterLab` to upload the model code constructed in section 3.1 and the preprocessed data from section 2.2.10 to the cloud container instance. Next, install the `sklearn` and `pandas` libraries required for the model on the cloud server. Since the server configuration already includes the `Tensorflow` library, there is no need to install it.

```
root@autodl-container-b52c468700-fbf4a369:~/model# pip install sklearn
Looking in indexes: http://mirrors.aliyun.com/pypi/simple
Collecting sklearn
  Downloading http://mirrors.aliyun.com/pypi/packages/46/1c/395a83ee7b2d2ad7a05b453872053d41449564477c81dc356f720b16eac4/sklearn-0.0.post12.tar.gz (2.6 kB)
```




## 3.3 Model Training Result

We split a large proportion of the training set from all preprocessed and format-standardized data using the `train_test_split` function from the `sklearn` library, and used these training sets to train the aforementioned models.

```
from sklearn.model_selection import train_test_split

self.ts_X_train, self.ts_X_test, self.static_X_train, self.static_X_test,
self.y_train, self.y_test = train_test_split(
            ts_X, static_X, y, test_size=0.2, random_state=42)
```

After 5-7 rounds of fine-tuning and approximately twenty minutes of training on the cloud service following each fine-tuning, we obtained a model capable of effectively predicting subsequent blood glucose concentration levels based on time-series data. We named this model `GCM_model.h5`, exported it, and saved it locally for subsequent predictions and performance evaluations.

# 4 Model Evaluation

## 4.1 Evaluation Based on Mean Absolute Error (MAE)

### 4.1.1 Evaluation Model Introduction

Mean Absolute Error (MAE) is a commonly used method for measuring the error of prediction models. `MAE` calculates the average of the absolute differences between predicted and actual values. The advantage of `MAE` is that it is simple and easy to understand, and it is not sensitive to outliers. Since `MAE` simply calculates the average of absolute errors, the contribution of each error is linear, making `MAE` more reflective of the actual level of prediction error. The smaller the `MAE` value, the stronger the predictive ability of the model.

### 4.1.2 Evaluation Process

We still used the `train_test_split` function from the `sklearn` library to split a small proportion of the test set and fed the test set into the already trained blood glucose concentration prediction model `GCM_model.h5`. We designed a method to use the MAE model to evaluate the error level between the blood glucose predictions made by `GCM_model.h5` based on the test set and the actual blood glucose values of the test set.

```
def evaluate_model(self):
        test_loss, test_mae = self.model.evaluate([self.ts_X_test,
self.static_X_test], self.y_test)
        print(f'Test loss: {test_loss}, Test MAE: {test_mae}')
        return f'Test loss: {test_loss}, Test MAE: {test_mae}'
```

We recorded the `loss` and `mae` of each epoch prediction compared to the actual values in `model_info.log` and exported and saved them for subsequent visual evaluation of the model's predictive ability. We also output the first 5 predicted values and the first 5 actual values directly in the console to quickly identify any serious errors in the model and decide whether to continue tuning.

```
with open("model_info.log", 'w') as file:
        # Print the loss and mae of each epoch
        for epoch, (loss, mae) in enumerate(zip(history.history['loss'],
history.history['mean_absolute_error'])):
            print(f"Epoch {epoch + 1}: Loss = {loss}, MAE = {mae}")
            file.write(f"Epoch {epoch + 1}: Loss = {loss}, MAE = {mae}\n")
        file.write("Evaluate Result: " + evaluate_result + '\n')

# Print prediction results
print(f'Predictions: {y_pred[:5]}')  # Display only the first 5 predictions
print(f'Actual: {y_test[:5]}')  # Display only the first 5 actual values
```

### 4.1.3 Evaluation Results & Visualization

The MAE-based evaluation results of this model saved in `model_info.log` are as follows:

```
Epoch 1: Loss = 0.24813230335712433, MAE = 0.3392144441604614
Epoch 2: Loss = 0.17603440582752228, MAE = 0.28965243697166443
Epoch 3: Loss = 0.16945107281208038, MAE = 0.28416118025779724
Epoch 4: Loss = 0.16450119018554688, MAE = 0.2798997163772583
Epoch 5: Loss = 0.1617734283208847, MAE = 0.2778536379337311
Epoch 6: Loss = 0.15967202186584473, MAE = 0.2754196226596832
Epoch 7: Loss = 0.15808287262916565, MAE = 0.27424079179763794
Epoch 8: Loss = 0.15543019771575928, MAE = 0.2722415626049042
Epoch 9: Loss = 0.152229905128479, MAE = 0.2698534429073334
Epoch 10: Loss = 0.15068094432353973, MAE = 0.2676985263824463
Epoch 11: Loss = 0.1485109180212021, MAE = 0.2659306228160858
Epoch 12: Loss = 0.14616045355796814, MAE = 0.264129638671875
Epoch 13: Loss = 0.14481528103351593, MAE = 0.2629924714565277
Epoch 14: Loss = 0.1432819664478302, MAE = 0.2612338960170746
Epoch 15: Loss = 0.14213888347148895, MAE = 0.26077672839164734
Epoch 16: Loss = 0.1405232697725296, MAE = 0.2589830756187439
Epoch 17: Loss = 0.13920897245407104, MAE = 0.2579362392425537
Epoch 18: Loss = 0.13700328767299652, MAE = 0.2558068633079529
Epoch 19: Loss = 0.135841503739357, MAE = 0.25503242015838623
Epoch 20: Loss = 0.1343383491039276, MAE = 0.25366151332855225
Epoch 21: Loss = 0.13319242000579834, MAE = 0.25238853693008423
```

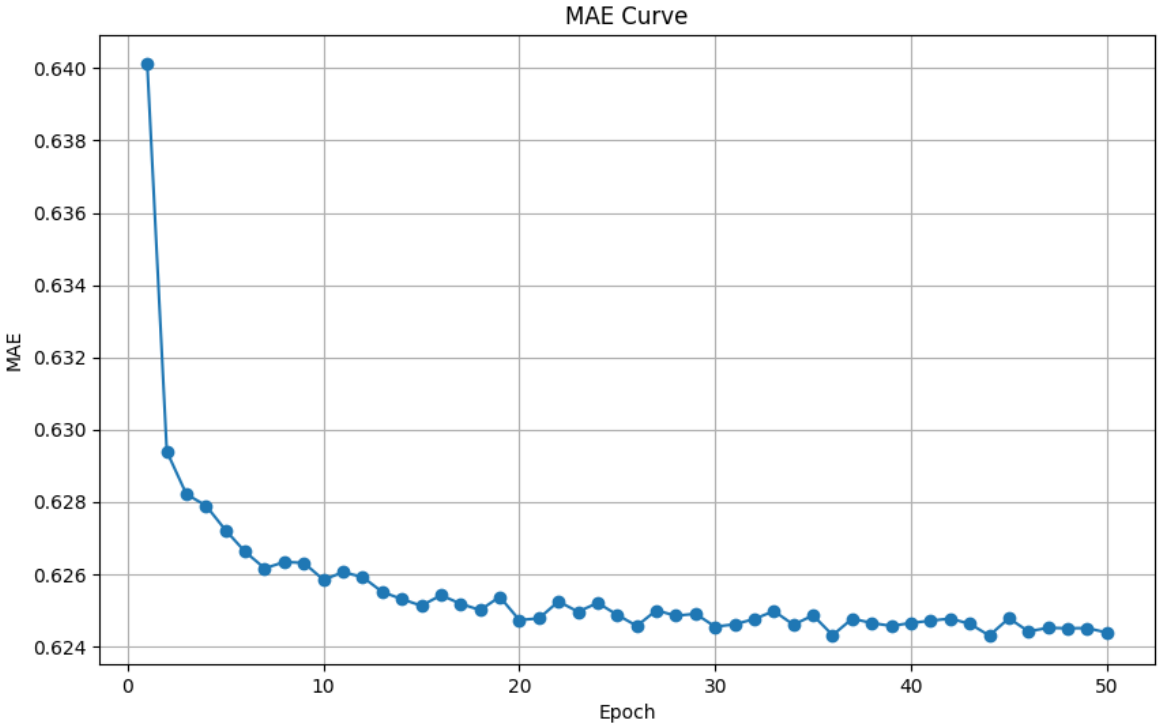
```
Epoch 22: Loss = 0.13157425820827484, MAE = 0.25120872259140015
Epoch 23: Loss = 0.13002479076385498, MAE = 0.24965637922286987
Epoch 24: Loss = 0.12899799644947052, MAE = 0.2487075924873352
Epoch 25: Loss = 0.12715759873390198, MAE = 0.24738220870494843
Epoch 26: Loss = 0.1254996806383133, MAE = 0.24577780067920685
Epoch 27: Loss = 0.12392573058605194, MAE = 0.24451224505901337
Epoch 28: Loss = 0.12294596433639526, MAE = 0.2437702715396881
Epoch 29: Loss = 0.12211327999830246, MAE = 0.24265168607234955
Epoch 30: Loss = 0.11990866810083389, MAE = 0.2406342476606369
Epoch 31: Loss = 0.11868849396705627, MAE = 0.23954859375953674
Epoch 32: Loss = 0.11719253659248352, MAE = 0.2382337599992752
Epoch 33: Loss = 0.11625178158283234, MAE = 0.23720043897628784
Epoch 34: Loss = 0.11457919329404831, MAE = 0.23559845983982086
Epoch 35: Loss = 0.11278732120990753, MAE = 0.23400256037712097
Epoch 36: Loss = 0.111522376537323, MAE = 0.23233428597450256
Epoch 37: Loss = 0.11087394505739212, MAE = 0.23186525702476501
Epoch 38: Loss = 0.10904955863952637, MAE = 0.23011435568332672
Epoch 39: Loss = 0.10804162174463272, MAE = 0.22891177237033844
Epoch 40: Loss = 0.1072082668542862, MAE = 0.22841022908687592
Epoch 41: Loss = 0.10584983229637146, MAE = 0.22709250450134277
Epoch 42: Loss = 0.1048450917005539, MAE = 0.22595353424549103
Epoch 43: Loss = 0.1035393550992012, MAE = 0.2244720607995987
Epoch 44: Loss = 0.10208194702863693, MAE = 0.22328290343284607
Epoch 45: Loss = 0.10101437568664551, MAE = 0.22216375172138214
Epoch 46: Loss = 0.09999130666255951, MAE = 0.22099186480045319
Epoch 47: Loss = 0.09851470589637756, MAE = 0.21978960931301117
Epoch 48: Loss = 0.09900260716676712, MAE = 0.22022844851016998
Epoch 49: Loss = 0.09660875797271729, MAE = 0.21708492934703827
Epoch 50: Loss = 0.0960533395409584, MAE = 0.2172061800956726
Evaluate Result: Test loss: 0.143349751830101, Test MAE: 0.2549135386943817
```

It is evident that the evaluated `loss` and `MAE` are both at relatively low levels, indicating that the blood glucose prediction ability of the `GCM_model.h5` model is strong and has met the project's expected goals.

To more intuitively display the blood glucose prediction capabilities of the `GCM_model.h5` model and the fluctuation range of its `MAE`, we visualized it using the `matplotlib.pyplot` library in `MAE_curve.py`:

```
plt.figure(figsize=(10, 6))
plt.plot(epochs, mae, marker='o', linestyle='-')
plt.title('MAE Curve')
plt.xlabel('Epoch')
plt.ylabel('MAE')
plt.grid(True)
plt.show()
```

The visualization of the evaluation results is as follows:




Since this curve was generated during the training process and the data used during training was standardized as described in section 3.3, some errors were inevitable. We eventually recalculated the `MAE` level using the original data. For the specific implementation process, please refer to section 4.3.3 The results are as follows:

```
------------------------------------
MSE for each time period:

MSE for 15 min: 205.9947820074595
MSE for 30 min: 338.8109404140507
MSE for 45 min: 478.94912597613234
MSE for 60 min: 632.00451303077 56
MSE for all: 413.9398403571045
------------------------------------
MAE for each time period:

MAE for 15 min: 9.773523023899 32
MAE for 30 min: 12.603488834097213
MAE for 45 min: 15.052373587510216
MAE for 60 min: 17.370290770187 47
MAE for all: 13.699919053923555
```

## 4.2 Evaluation Based on Predictions of the Entire Dataset

### 4.2.1 Evaluation Process

To minimize the impact of extreme data and increase the test data base while facilitating the generation of comprehensive visualization results, we used the entire dataset as the test set in `model_test.py`, feeding it into the previously trained `GCM_model.h5` model. We then compared the prediction data provided by the model with the actual values of the entire dataset for subsequent visualization.

```
model = TimeModel()
model.load_model("GCM_model.h5")
y_pred, y_test = model.predict()
with open("y_pred.json", "w") as file:
    y_pred_str = json.dumps(y_pred.tolist(), indent=4)
    file.write(y_pred_str)
with open("y_test.json", "w") as file:
    y_test_str = json.dumps(y_test.tolist(), indent=4)
    file.write(y_test_str)
```

### 4.2.2 Evaluation Results & Visualization

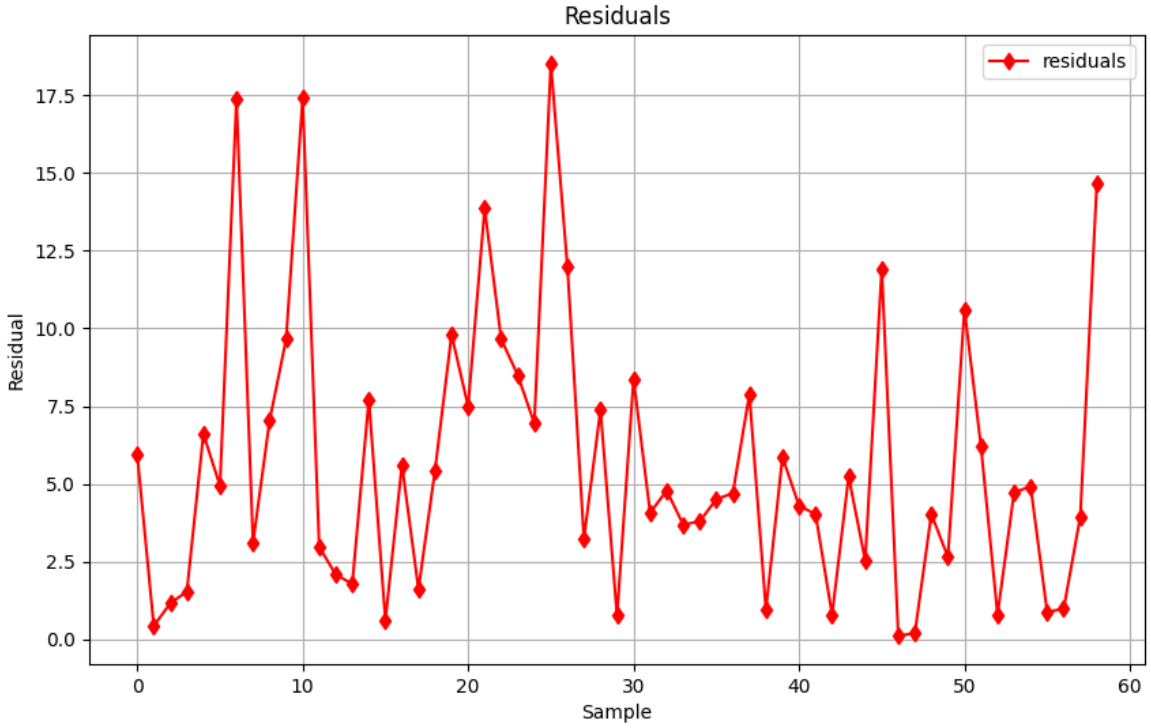
We exported all predicted values and saved them in `y_pred.json` in the local project. All actual values were exported and saved in `y_test.json` in the local project. In `pred_real_compare.py`, we analyze the residuals and residual percentages between the predicted values and the actual values using the following formula:

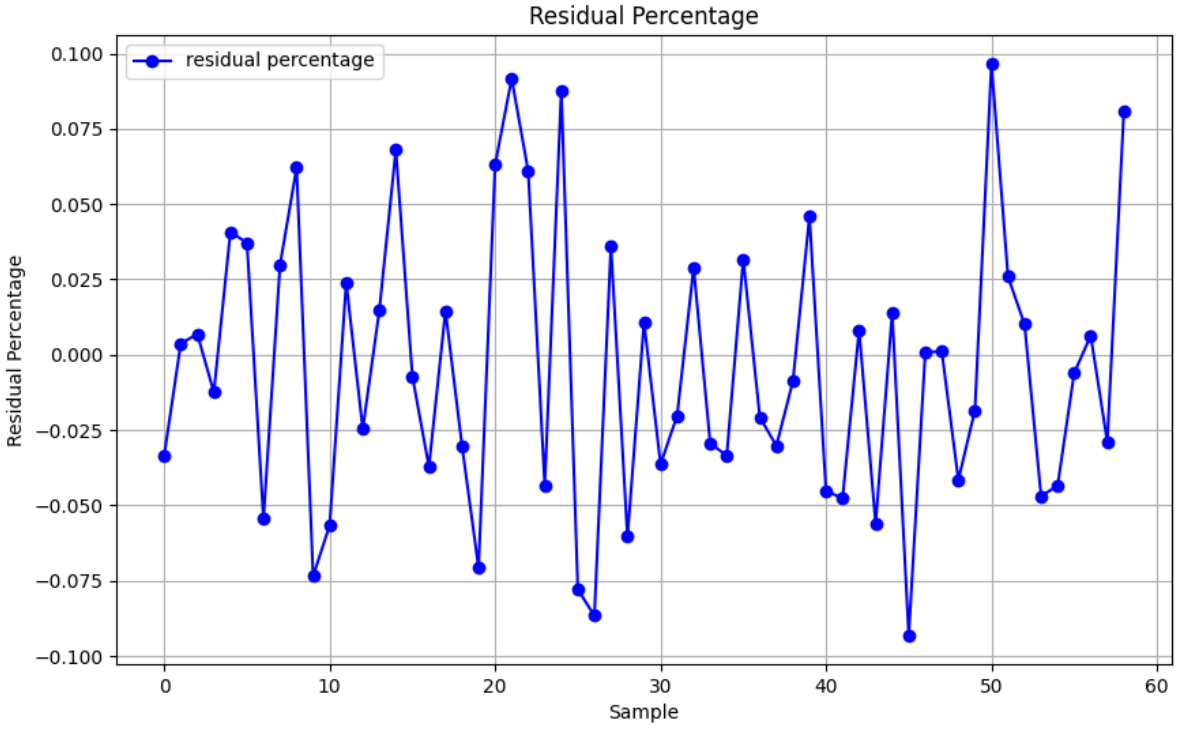
```
residual_percentage = [(t - p) / t * 100 if t != 0 else 0 for t, p in
zip(test_values_sampled, pred_values_sampled)]

filtered_residuals = []
filtered_residual_percentage = []
for residual, percentage in zip(residuals_sampled, residual_percentage):
    if residual <= 20 and percentage <= 10 and percentage >= -10:
        filtered_residuals.append(residual)
        filtered_residual_percentage.append(percentage/100)
```

Afterward, we plot the residuals and residual percentage graphs:

```
# Plotting the residuals
plt.figure(figsize=(10, 6))
plt.plot(range(len(filtered_residuals)), filtered_residuals, label='residuals',
marker='d', color='r', linestyle='-')
plt.xlabel('Sample')
plt.ylabel('Residual')
plt.title('Residuals')
plt.legend()
plt.grid(True)
plt.show()

# Plotting the residual percentage
plt.figure(figsize=(10, 6))
plt.plot(range(len(filtered_residual_percentage)), filtered_residual_percentage,
label='residual percentage', marker='o', color='b', linestyle='-')
plt.xlabel('Sample')
plt.ylabel('Residual Percentage')
plt.title('Residual Percentage')
plt.legend()
plt.grid(True)
plt.show()
```

The visualization results are as follows:




## 4.3 Evaluation Based on Mean Squared Error (MSE)

### 4.3.1 Introduction to the Evaluation Model

Mean Squared Error (MSE) is a commonly used metric for evaluating the performance of prediction models. It measures the model's error by calculating the average of the squared differences between the predicted values and the actual values. Since the errors are squared, `MSE` assigns greater weight to larger errors, making it particularly useful for identifying models that occasionally produce large deviations. `MSE` is especially

suitable for evaluating regression models, providing a clear quantitative measure of the model's predictive accuracy.

### 4.3.2 Evaluation Process

First, in `cal_MSE_MAE.py`, we read the `y_pred.json` data produced by the previous model and the original `y_test.json` data, and initialize the lists.

```
test_values = []
pred_values = []

with open(TEST_PATH, 'r') as file:
    test_values = json.load(file)

with open(PRED_PATH, 'r') as file:
    pred_values = json.load(file)

# Initialize lists
test_15min, test_30min, test_45min, test_60min = [], [], [], []
pred_15min, pred_30min, pred_45min, pred_60min = [], [], [], []
```

Then, we split the blood glucose concentration data for 15, 30, 45, and 60 minutes:

```
for i in range(len(test_values)):
    test_15min.append(test_values[i][0])
    test_30min.append(test_values[i][1])
    test_45min.append(test_values[i][2])
    test_60min.append(test_values[i][3])

    pred_15min.append(pred_values[i][0])
    pred_30min.append(pred_values[i][1])
    pred_45min.append(pred_values[i][2])
    pred_60min.append(pred_values[i][3])
```

Finally, we calculate the `MSE` levels and print the results:

```
mse_15min = np.mean((np.array(test_15min) - np.array(pred_15min)) ** 2)
mse_30min = np.mean((np.array(test_30min) - np.array(pred_30min)) ** 2)
mse_45min = np.mean((np.array(test_45min) - np.array(pred_45min)) ** 2)
mse_60min = np.mean((np.array(test_60min) - np.array(pred_60min)) ** 2)
```

### 4.3.3 Evaluation Results & Visualization

The evaluation results based on Mean Squared Error (MSE) are as follows:

```
-------------------------------------
MSE for each time period:

MSE for 15 min: 205.9947820074595
MSE for 30 min: 338.8109404140507
MSE for 45 min: 478.94912597613234
MSE for 60 min: 632.00451303077 56
MSE for all: 413.9398403571045
-------------------------------------
MAE for each time period:

MAE for 15 min: 9.773523023899 32
MAE for 30 min: 12.603488834097213
MAE for 45 min: 15.052373587510216
MAE for 60 min: 17.370290770187 47
MAE for all: 13.699919053923555
```