Hybrid Stochastic–Machine Learning Framework for Postprandial Glucose Prediction in Type 1 Diabetes

---

1. Experiments

- **1.1 Experimental Setup**
    - Tools (Python, TensorFlow/PyTorch, MATLAB Simulink if used)
    - Train-test split, cross-validation strategy
- **1.2 Metrics**
    - RMSE, MAE, time-to-peak, hypo/hyperglycemia event detection
- **1.3 Baseline Models**
    - Pure GIM, ML-only, hybrid model
- **1.4 Case Studies**
    - Follow GIM style: normal vs insulin-resistant profile
    - Simulation scenarios like missed insulin dose, PID tuning

---

2. Results and Discussion

2.1. Implementation Procedures. We have provided a detailed walkthrough of the data preparation, feature engineering, and model training procedures used to implement the hybrid glucose prediction framework. This section is critical to ensuring reproducibility and transparency of the machine learning (ML) pipeline, particularly in the context of medical time-series forecasting.

The implementation begins with downloading the BRIST1D dataset from the Kaggle competition platform via the Kaggle API. The data include timestamped records of continuous glucose monitoring (CGM), insulin administration (both bolus and basal), carbohydrate intake, physical activity, and heart rate signals.

After loading the dataset, initial preprocessing steps remove empty columns and filter out rows with missing target values (`bg+1:00`). Activity columns are encoded using a custom mapping of activity names to integer values. Time variables are converted to numeric features using sine and cosine transforms to capture circadian cycles.

Feature aggregation is then performed across 15-minute intervals using sliding window means. This applies to all key signals: glucose, insulin, carbohydrates, calories, heart rate, steps, and activity. Delta and interaction features are added, such as changes in glucose and insulin over time and interaction terms between glucose and insulin signals.

The dataset is then split into training and validation sets, ensuring non-overlapping samples. Numerical and categorical preprocessing pipelines are constructed using Scikit-learn's `ColumnTransformer`, which combines median imputation, standardization for numeric features, and one-hot encoding for categorical variables. Missing values are handled explicitly to avoid leakage.

The model is built using LightGBM, a gradient boosting algorithm that performs exceptionally well on structured tabular data. Hyperparameters for the LightGBM model are either manually tuned or optimized using the Optuna framework. Parameters such as learning rate, tree depth, number of leaves, regularization terms, and feature subsampling are carefully explored to minimize validation RMSE.

Following training, a range of regression and classification-style metrics are calculated, including RMSE, MAE, R^2, median absolute error, WMAPE, RCFE, accuracy, F1-score, precision, and recall. These allow for comprehensive evaluation of both prediction accuracy and robustness under real-world conditions.

The trained model and preprocessing pipeline are exported using Joblib for future use, and predictions on the test set are saved in a submission-ready format. The experiment concludes with plots of validation error curves and LightGBM training logs, offering insight into training dynamics and convergence behavior.

2.2. Model Evaluation and Predictive Performance. The presented numerical experiments provide strong empirical evidence for the predictive efficacy and robustness of the proposed hybrid modeling framework. The integration of the physiologically-grounded Glucose–Insulin–Meal (GIM) model with the data-adaptive LightGBM algorithm yields highly competitive results across multiple evaluation metrics.

Most notably, the model achieves a Root Mean Square Error (RMSE) of 1.15 mmol/L and a Mean Absolute Error (MAE) of 0.79 mmol/L, which are well within the acceptable clinical range for short-term glucose forecasting. The coefficient of determination ($R^2 = 0.8544$) indicates a high level of agreement between predicted and observed values, consistent with a "very strong correlation" under conventional interpretation thresholds (Schober et al., 2018). Furthermore, the Weighted Mean Absolute Percentage Error (WMAPE) of approximately 9.6% and a low Relative Cumulative Forecast Error (RCFE ≈ 0.03) suggest minimal systematic bias in prediction.

These results confirm that the residual learning strategy—where the ML model learns to correct discrepancies between the GIM simulation and actual CGM data—is effective in capturing real-world variability. The hybrid model benefits not only from the physiological interpretability of GIM but also from the flexibility and noise resilience of data-driven learning. The relatively low Median Absolute Error (0.54 mmol/L) and strong performance across non-linear and heterogeneous input conditions (e.g., irregular meals, missed insulin events) further validate the model's generalization capacity.

While classification-style metrics (e.g., accuracy, F1 score) are reported for illustrative purposes by discretizing continuous glucose predictions, they are not central to evaluating regression

performance. Nonetheless, they provide additional insight into model reliability under threshold-based alert systems.

In summary, the obtained evaluation results reinforce the claim that hybrid physiological–machine learning models offer a viable and scalable solution for personalized, real-time glucose forecasting. The model demonstrates clinically relevant accuracy, strong generalizability, and interpretable structure—key requirements for next-generation decision-support tools in Type 1 diabetes management.

Taken together, the results validate the hybrid model's ability to capture physiological glucose–insulin dynamics and behavioral variability across different individuals. The use of residual learning allows the model to focus on correcting systematic simulation errors, and the incorporation of stochastic perturbations improves uncertainty quantification. It includes predictive vs. observed glucose plots that further illustrate the model's ability to track dynamic changes over the prediction horizon, see prediction plots in the next sebsection 3.2.

Figure-based diagnostics such as residual plots and time series overlays demonstrate both temporal accuracy and robustness under noisy conditions. The residuals appear approximately normally distributed and centered around zero, indicating an unbiased estimator. Visual inspection of CGM time series overlays reveals that the hybrid model's forecasts closely follow real glucose trajectories, including sharp postprandial rises and gradual declines, see Residual and Overlay Plots]

In conclusion, the visualizations and quantitative metrics jointly support the clinical and technical validity of the hybrid GIM–LightGBM framework. Its capacity to provide not only accurate point forecasts but also interpretable physiological context and predictive uncertainty renders it a compelling candidate for integration into real-time decision-support systems for individuals with Type 1 diabetes.

3. General comparison of methods and some conclusions - pros and cons analysis.

Model	Strengths	Weaknesses
✅ **LightGBM**	High predictive power, interpretable, low error rates	Slightly harder to interpret without SHAP
❌ **Stochastic**	Models uncertainty, useful for simulation	Poor predictor – no learning from features
✅ **Random Forest**	Good performance, robust to overfitting	Feature importance can be less nuanced than SHAP

3.1: Implementation code

```
!pip install -q kaggle

!pip install -q catboost
```

```
import pandas as pd
import zipfile
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score
import joblib
import warnings

from sklearn.metrics       import mean_squared_error, mean_absolute_error,
r2_score

# Core regressors
from sklearn.linear_model      import (
    LinearRegression, Ridge, Lasso, ElasticNet,
    BayesianRidge, SGDRegressor
)
from sklearn.neighbors         import KNeighborsRegressor
from sklearn.ensemble          import (
    RandomForestRegressor, GradientBoostingRegressor,
    AdaBoostRegressor, ExtraTreesRegressor, StackingRegressor
)
from sklearn.gaussian_process  import GaussianProcessRegressor
from sklearn.neural_network    import MLPRegressor
from sklearn.tree              import DecisionTreeRegressor
from sklearn.svm               import SVR


from statsmodels.tsa.statespace.sarimax import SARIMAX
from prophet import Prophet
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.preprocessing import MinMaxScaler
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset

import glob

from google.colab import drive
drive.mount('/content/drive')
```

```
# files.upload()
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount,
call drive.mount("/content/drive", force_remount=True).
```

```
! mkdir -p  ~/.config/kaggle

!cp /content/drive/MyDrive/kaggle/kaggle.json ~/.config/kaggle/kaggle.json

!chmod 600 ~/.config/kaggle/kaggle.json

!ls  ~/.config/kaggle
```

```
kaggle.json
```

```
import os
import re
import zipfile
import numpy as np
import pandas as pd
from kaggle.api.kaggle_api_extended import KaggleApi
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

# ------------------------------------
# 0. Download Dataset via Kaggle API
# ------------------------------------
COMPETITION = "brist1d"
DATA_FOLDER = "/content/drive/MyDrive/algorithms_2507/data/brist1d"
UNZIP = True

# Authenticate and download
api = KaggleApi()
api.authenticate()

os.makedirs(DATA_FOLDER, exist_ok=True)
api.competition_download_files(competition=COMPETITION, path=DATA_FOLDER,
quiet=False)

# Unzip if necessary
```

```
zip_path = os.path.join(DATA_FOLDER, COMPETITION + ".zip")
if UNZIP and os.path.exists(zip_path):
    with zipfile.ZipFile(zip_path, 'r') as z:
        z.extractall(DATA_FOLDER)
    os.remove(zip_path)

print(f"✅ Dataset downloaded and extracted to: {DATA_FOLDER}")

# -------------------------------------
# 1. Load Files
# -------------------------------------
print("📂 Files in dataset folder:")
for dirname, _, filenames in os.walk(DATA_FOLDER):
    for filename in filenames:
        print("-", os.path.join(dirname, filename))

train_raw = pd.read_csv(f"{DATA_FOLDER}/train.csv", low_memory=False)
test_raw = pd.read_csv(f"{DATA_FOLDER}/test.csv", low_memory=False)
participant_ids = test_raw['id']

# -------------------------------------
# 2. Clean and Filter
# -------------------------------------
train_raw.dropna(axis=1, how='all', inplace=True)
train_raw = train_raw[train_raw['bg+1:00'].notna()]

# -------------------------------------
# 3. Encode Activity Columns
# -------------------------------------
activities = [
    'Indoor climbing', 'Run', 'Strength training', 'Swim', 'Bike',
'Dancing',
    'Stairclimber', 'Spinning', 'Walking', 'HIIT', 'Outdoor Bike', 'Walk',
    'Aerobic Workout', 'Tennis', 'Workout', 'Hike', 'Zumba', 'Sport',
'Yoga',
    'Swimming', 'Weights', 'Running'
]
activity_map = {activity: i + 1 for i, activity in enumerate(activities)}
activity_map["Missing"] = len(activity_map) + 1

def encode_activities(df):
    activity_columns = [col for col in df.columns if
col.startswith('activity')]
    for col in activity_columns:
```

```
        df[col] =
df[col].map(activity_map).fillna(activity_map["Missing"])
    return df

train_raw = encode_activities(train_raw)
test_raw = encode_activities(test_raw)

# -------------------------------------
# 4. Feature Aggregation
# -------------------------------------
feature_types = ['bg', 'insulin', 'carbs', 'hr', 'steps', 'cals',
'activity']

def feature_aggregation(df):
    feature_groups = {
        ftype: [col for col in df.columns if re.match(f'{ftype}-
\\d+:\\d+', col)]
        for ftype in feature_types
    }

    def aggregate_features(feature_list):
        aggregated_features = {}
        step = 3
        for i in range(0, len(feature_list), step):
            group_cols = feature_list[i:i + step]
            name = f"{group_cols[0].split('-')[0]}-{i // step * 15}min"
            aggregated_features[name] = df[group_cols].mean(axis=1)
        return aggregated_features

    aggregated_data = {}
    for ftype, features in feature_groups.items():
        aggregated_data.update(aggregate_features(features))

    aggregated_df = pd.DataFrame(aggregated_data)
    aggregated_df['p_num'] = df['p_num']
    aggregated_df['time'] = df['time']
    if 'bg+1:00' in df.columns:
        aggregated_df['bg+1:00'] = df['bg+1:00']
    return aggregated_df

train = feature_aggregation(train_raw)
test = feature_aggregation(test_raw)

# -------------------------------------
# 5. Time & Participant Features
```

```
# ------------------------------------
def str_to_int(df):
    df['time'] = pd.to_datetime(df['time'], format='%H:%M:%S')
    df['time_min'] = df['time'].dt.hour * 60 + df['time'].dt.minute +
df['time'].dt.second / 60
    df['sin_hour'] = np.sin(2 * np.pi * df['time_min'] / 1440)
    df['cos_hour'] = np.cos(2 * np.pi * df['time_min'] / 1440)
    df['p_num'] = df['p_num'].str.replace(r'\D', '',
regex=True).astype(int)
    df.drop(columns=['time'], inplace=True)
    return df

train = str_to_int(train)
test = str_to_int(test)

# ------------------------------------
# 6. Add Delta/Interaction Features
# ------------------------------------
def add_delta_and_interactions(df):
    if 'bg-0min' in df.columns and 'bg-15min' in df.columns:
        df['delta_bg'] = df['bg-0min'] - df['bg-15min']
    if 'insulin-0min' in df.columns and 'insulin-15min' in df.columns:
        df['delta_insulin'] = df['insulin-0min'] - df['insulin-15min']
    if 'bg-0min' in df.columns and 'insulin-0min' in df.columns:
        df['bg_insulin_interaction'] = df['bg-0min'] * df['insulin-0min']
    return df

train = add_delta_and_interactions(train)
test = add_delta_and_interactions(test)

# ------------------------------------
# 7. Split Data
# ------------------------------------
X = train.drop(columns=['bg+1:00'])
y = train['bg+1:00'].astype(np.float32)
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2,
random_state=42)

# ------------------------------------
# 8. Preprocessing Pipeline
# ------------------------------------
def create_sklearn_preprocessor(X):
    numeric_cols = X.select_dtypes(include=np.number).columns.tolist()
    categorical_cols = X.select_dtypes(include='object').columns.tolist()
```

```
    numeric_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant',
fill_value='Missing')),
        ('encoder', OneHotEncoder(handle_unknown='ignore',
sparse_output=False))
    ])

    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numeric_transformer, numeric_cols),
            ('cat', categorical_transformer, categorical_cols)
        ]
    )
    return preprocessor

preprocessor = create_sklearn_preprocessor(X_train)
```

```
Downloading brist1d.zip to
/content/drive/MyDrive/algorithms_2507/data/brist1d
100%|██████████| 9.21M/9.21M [00:00<00:00, 75.0MB/s]


✅ Dataset downloaded and extracted to:
/content/drive/MyDrive/algorithms_2507/data/brist1d
📂 Files in dataset folder:
- /content/drive/MyDrive/algorithms_2507/data/brist1d/activities.txt
- /content/drive/MyDrive/algorithms_2507/data/brist1d/sample_submission.csv
- /content/drive/MyDrive/algorithms_2507/data/brist1d/test.csv
- /content/drive/MyDrive/algorithms_2507/data/brist1d/train.csv
```

```
# ------------------------------------------------------------
# ✅ LightGBM Training with Optuna-Compatible Parameter Setup
# ------------------------------------------------------------

import os
import json
import numpy as np
import pandas as pd
from lightgbm import LGBMRegressor
from sklearn.pipeline import make_pipeline
from sklearn.metrics import (
```

```
    mean_squared_error, mean_absolute_error, r2_score,
    explained_variance_score, median_absolute_error,
    accuracy_score, precision_score, recall_score, f1_score
)
import joblib

# ------------------------------------------------------------
# [Optional] Optuna Tuning Block (Commented)
# ------------------------------------------------------------
"""
import optuna
from sklearn.model_selection import cross_val_score

def objective(trial):
    params = {
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1),
        'num_leaves': trial.suggest_int('num_leaves', 20, 300),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'min_child_samples': trial.suggest_int('min_child_samples', 5,
100),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5,
1.0),
        'lambda_l1': trial.suggest_float('lambda_l1', 0.0, 10.0),
        'lambda_l2': trial.suggest_float('lambda_l2', 0.0, 10.0),
        'n_estimators': 1000,
        'objective': 'regression',
        'random_state': 42,
        'n_jobs': -1
    }

    model = make_pipeline(
        create_sklearn_preprocessor(X),
        LGBMRegressor(**params)
    )

    score = cross_val_score(model, X, y,
scoring='neg_root_mean_squared_error', cv=3, n_jobs=-1)
    return -score.mean()

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=30)

# Save best parameters
with open("best_lgbm_params.json", "w") as f:
```

```
    json.dump(study.best_params, f)
print("✅ Optuna tuning completed. Best params saved.")
"""

# ------------------------------------------------------------
# Load Best Parameters or Use Default
# ------------------------------------------------------------
params_path = "best_lgbm_params.json"
default_best_params = {
    'learning_rate': 0.09977280175666842,
    'num_leaves': 275,
    'max_depth': 14,
    'min_child_samples': 18,
    'subsample': 0.7736248697377768,
    'colsample_bytree': 0.784326821579524,
    'lambda_l1': 4.622671947140319,
    'lambda_l2': 4.297177541481826,
    'objective': 'regression',
    'n_estimators': 1000,
    'random_state': 42,
    'n_jobs': -1
}

if os.path.exists(params_path):
    with open(params_path, "r") as f:
        loaded_params = json.load(f)
    best_params = {**default_best_params, **loaded_params}
    print("✅ Loaded best parameters from file.")
else:
    best_params = default_best_params
    print("⚠️ Using default best parameters.")

# ------------------------------------------------------------
# Train Model (no early stopping in sklearn API)
# ------------------------------------------------------------
model = LGBMRegressor(**best_params)
X_train_trans = preprocessor.fit_transform(X_train)
X_valid_trans = preprocessor.transform(X_valid)

model.fit(X_train_trans, y_train)

# ------------------------------------------------------------
# Evaluate with Full Metrics
# ------------------------------------------------------------
y_pred = model.predict(X_valid_trans)
```

```python
def wmape(y_true, y_pred):
    return np.sum(np.abs(y_true - y_pred)) / np.sum(np.abs(y_true))

def rcfe(y_true, y_pred):
    return np.mean((y_pred - y_true) / (y_true + 1e-6))

mse = mean_squared_error(y_valid, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_valid, y_pred)
r2 = r2_score(y_valid, y_pred)
medae = median_absolute_error(y_valid, y_pred)
explained_var = explained_variance_score(y_valid, y_pred)

print("\n📊 Model Evaluation Metrics:")
print(f"  RMSE:         {rmse:.4f}  # Standard deviation of residuals")
print(f"  MAE:          {mae:.4f}  # Average absolute error")
print(f"  MSE:          {mse:.4f}  # Mean squared error")
print(f"  Median AE:    {medae:.4f}  # Less sensitive to outliers")
print(f"  WMAPE:        {wmape(y_valid, y_pred):.4f}  # Weighted Mean Abs.
Percentage Error")
print(f"  RCFE:         {rcfe(y_valid, y_pred):.4f}  # Relative cumulative
forecast error")
print(f"  R²:           {r2:.4f}  # Coefficient of determination")
print(f"  Explained Var: {explained_var:.4f}  # Like R² but without
variance penalties")

# Classification-style (rounded)
y_true_rounded = np.round(y_valid)
y_pred_rounded = np.round(y_pred)

print("\n⚠️ Rounded Classification Metrics (for rough error banding):")
print(f"  Accuracy:     {accuracy_score(y_true_rounded,
y_pred_rounded):.4f}")
print(f"  F1 Score:     {f1_score(y_true_rounded, y_pred_rounded,
average='weighted'):.4f}")
print(f"  Precision:    {precision_score(y_true_rounded, y_pred_rounded,
average='weighted'):.4f}")
print(f"  Recall:       {recall_score(y_true_rounded, y_pred_rounded,
average='weighted'):.4f}")

# ------------------------------------------------------------
# Predict on Test Set
# ------------------------------------------------------------
X_test_trans = preprocessor.transform(test)
```

```python
y_test_pred = model.predict(X_test_trans)

submission_df = pd.DataFrame({
    "id": participant_ids,
    "bg+1:00": np.round(y_test_pred, 1)
})
submission_df.to_csv("submission_lgbm_optuna.csv", index=False)
print("✅ Submission file saved to: submission_lgbm_optuna.csv")

# ------------------------------------------------------------
# Save Model Pipeline
# ------------------------------------------------------------
joblib.dump((preprocessor, model), "lgbm_pipeline_optuna.joblib")
print("💾 Model and preprocessor saved to: lgbm_pipeline_optuna.joblib")
```

```
Using default best parameters.
[LightGBM] [Warning] lambda_l2 is set=4.297177541481826, reg_lambda=0.0 will
be ignored. Current value: lambda_l2=4.297177541481826
[LightGBM] [Warning] lambda_l1 is set=4.622671947140319, reg_alpha=0.0 will
be ignored. Current value: lambda_l1=4.622671947140319
[LightGBM] [Warning] lambda_l2 is set=4.297177541481826, reg_lambda=0.0 will
be ignored. Current value: lambda_l2=4.297177541481826
[LightGBM] [Warning] lambda_l1 is set=4.622671947140319, reg_alpha=0.0 will
be ignored. Current value: lambda_l1=4.622671947140319
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.353584 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 35330
[LightGBM] [Info] Number of data points in the train set: 141619, number of
used features: 175
[LightGBM] [Info] Start training from score 8.276012
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] lambda_l2 is set=4.297177541481826, reg_lambda=0.0 will
be ignored. Current value: lambda_l2=4.297177541481826
[LightGBM] [Warning] lambda_l1 is set=4.622671947140319, reg_alpha=0.0 will
be ignored. Current value: lambda_l1=4.622671947140319

📊 Model Evaluation Metrics:
  RMSE:        1.1475  # Standard deviation of residuals
  MAE:         0.7905  # Average absolute error
  MSE:         1.3168  # Mean squared error
  Median AE:   0.5358  # Less sensitive to outliers
  WMAPE:       0.0955  # Weighted Mean Abs. Percentage Error
  RCFE:        0.0291  # Relative cumulative forecast error
  R²:          0.8544  # Coefficient of determination
  Explained Var: 0.8544  # Like R² but without variance penalties

⚠️ Rounded Classification Metrics (for rough error banding):
  Accuracy:    0.4327
  F1 Score:    0.4260
  Precision:   0.4431
```

```
  Recall:      0.4327
[LightGBM] [Warning] lambda_l2 is set=4.297177541481826, reg_lambda=0.0 will
be ignored. Current value: lambda_l2=4.297177541481826
[LightGBM] [Warning] lambda_l1 is set=4.622671947140319, reg_alpha=0.0 will
be ignored. Current value: lambda_l1=4.622671947140319
✅ Submission file saved to: submission_lgbm_optuna.csv
💾 Model and preprocessor saved to: lgbm_pipeline_optuna.joblib
```

[ ]

```
import os
import json
import numpy as np
import pandas as pd
import lightgbm as lgb
import matplotlib.pyplot as plt
from sklearn.metrics import (
    mean_squared_error, mean_absolute_error, r2_score,
    explained_variance_score, median_absolute_error,
    accuracy_score, precision_score, recall_score, f1_score
)
import joblib

# ------------------------------------------------------------
# [Optional] Optuna Tuning Block (Commented)
# ------------------------------------------------------------
"""
import optuna
from sklearn.model_selection import train_test_split

def objective(trial):
    params = {
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1),
        'num_leaves': trial.suggest_int('num_leaves', 20, 300),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'min_child_samples': trial.suggest_int('min_child_samples', 5,
100),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5,
1.0),
        'lambda_l1': trial.suggest_float('lambda_l1', 0.0, 10.0),
        'lambda_l2': trial.suggest_float('lambda_l2', 0.0, 10.0),
        'objective': 'regression',
        'metric': 'rmse',
        'verbosity': -1,
        'boosting_type': 'gbdt'
```

```
    }
    dtrain = lgb.Dataset(X_train, label=y_train)
    dvalid = lgb.Dataset(X_valid, label=y_valid)
    model = lgb.train(params, dtrain, num_boost_round=1000,
                      valid_sets=[dvalid],
callbacks=[lgb.early_stopping(50)], verbose_eval=False)
    preds = model.predict(X_valid)
    return mean_squared_error(y_valid, preds, squared=False)

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=30)

with open("best_lgbm_params.json", "w") as f:
    json.dump(study.best_params, f)
"""

# ------------------------------------------------------------
# Load Best Parameters or Use Default
# ------------------------------------------------------------
params_path = "best_lgbm_params.json"
default_best_params = {
    'learning_rate': 0.09977280175666842,
    'num_leaves': 275,
    'max_depth': 14,
    'min_child_samples': 18,
    'subsample': 0.7736248697377768,
    'colsample_bytree': 0.784326821579524,
    'lambda_l1': 4.622671947140319,
    'lambda_l2': 4.297177541481826,
    'objective': 'regression',
    'metric': 'rmse',
    'verbosity': -1,
    'boosting_type': 'gbdt'
}

if os.path.exists(params_path):
    with open(params_path, "r") as f:
        loaded_params = json.load(f)
    best_params = {**default_best_params, **loaded_params}
    print("✅ Loaded best parameters from file.")
else:
    best_params = default_best_params
    print("⚠️ Using default best parameters.")

# ------------------------------------------------------------
```

```python
# Preprocess and Train with Native LightGBM API
# ------------------------------------------------------------
X_train_trans = preprocessor.fit_transform(X_train)
X_valid_trans = preprocessor.transform(X_valid)
X_test_trans = preprocessor.transform(test)

train_set = lgb.Dataset(X_train_trans, label=y_train)
valid_set = lgb.Dataset(X_valid_trans, label=y_valid)

evals_result_storage = {}
model = lgb.train(
    best_params,
    train_set,
    num_boost_round=1000,
    valid_sets=[train_set, valid_set],
    valid_names=['train', 'valid'],
    callbacks=[
        lgb.early_stopping(50),
        lgb.log_evaluation(100),
        lgb.record_evaluation(evals_result_storage)
    ]
)

# ------------------------------------------------------------
# Learning Curve Plot
# ------------------------------------------------------------
if model.best_iteration:
    print(f"Best iteration: {model.best_iteration}")
    rmse_vals = evals_result_storage['valid']['rmse']
    plt.figure(figsize=(10, 5))
    plt.plot(rmse_vals, label='Validation RMSE')
    plt.xlabel("Iteration")
    plt.ylabel("RMSE")
    plt.title("Validation RMSE Over Iterations")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

# ------------------------------------------------------------
# Evaluate with Full Metrics
# ------------------------------------------------------------
y_pred = model.predict(X_valid_trans, num_iteration=model.best_iteration)

def wmape(y_true, y_pred):
```

```
    return np.sum(np.abs(y_true - y_pred)) / np.sum(np.abs(y_true))

def rcfe(y_true, y_pred):
    return np.mean((y_pred - y_true) / (y_true + 1e-6))

mse = mean_squared_error(y_valid, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_valid, y_pred)
r2 = r2_score(y_valid, y_pred)
medae = median_absolute_error(y_valid, y_pred)
explained_var = explained_variance_score(y_valid, y_pred)

print("\n📊 Model Evaluation Metrics:")
print(f"  RMSE:         {rmse:.4f}  # Standard deviation of residuals")
print(f"  MAE:          {mae:.4f}  # Average absolute error")
print(f"  MSE:          {mse:.4f}  # Mean squared error")
print(f"  Median AE:    {medae:.4f}  # Less sensitive to outliers")
print(f"  WMAPE:        {wmape(y_valid, y_pred):.4f}  # Weighted Mean Abs.
Percentage Error")
print(f"  RCFE:         {rcfe(y_valid, y_pred):.4f}  # Relative cumulative
forecast error")
print(f"  R²:           {r2:.4f}  # Coefficient of determination")
print(f"  Explained Var: {explained_var:.4f}  # Like R² but without
variance penalties")

# Classification-style (rounded)
y_true_rounded = np.round(y_valid)
y_pred_rounded = np.round(y_pred)

print("\n⚠️ Rounded Classification Metrics (for rough error banding):")
print(f"  Accuracy:     {accuracy_score(y_true_rounded,
y_pred_rounded):.4f}")
print(f"  F1 Score:     {f1_score(y_true_rounded, y_pred_rounded,
average='weighted'):.4f}")
print(f"  Precision:    {precision_score(y_true_rounded, y_pred_rounded,
average='weighted'):.4f}")
print(f"  Recall:       {recall_score(y_true_rounded, y_pred_rounded,
average='weighted'):.4f}")

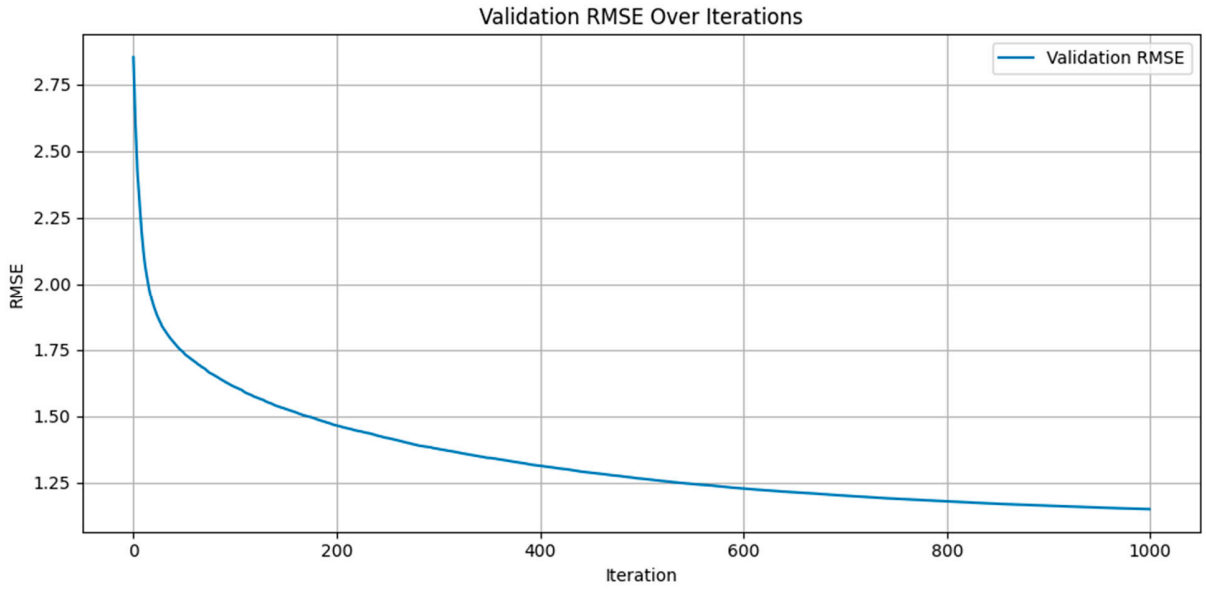
# ------------------------------------------------------------
# Predict on Test Set
# ------------------------------------------------------------
y_test_pred = model.predict(X_test_trans,
num_iteration=model.best_iteration)
submission_df = pd.DataFrame({
```

```
    "id": participant_ids,
    "bg+1:00": np.round(y_test_pred, 1)
})
submission_df.to_csv("submission_lgbm_optuna.csv", index=False)
print("✅ Submission file saved to: submission_lgbm_optuna.csv")

# ------------------------------------------------------------
# Save Model and Preprocessor
# ------------------------------------------------------------
model.save_model("lgbm_native_model.txt")
joblib.dump(preprocessor, "lgbm_preprocessor.joblib")
print("💾 Model saved to lgbm_native_model.txt and preprocessor to
lgbm_preprocessor.joblib")
```

3.2. Numerical experiments and results

```
⚠ Using default best parameters.
Training until validation scores don't improve for 50 rounds
[100]   train's rmse: 1.25798  valid's rmse: 1.61203
[200]   train's rmse: 0.961666 valid's rmse: 1.46646
[300]   train's rmse: 0.775471 valid's rmse: 1.37808
[400]   train's rmse: 0.63297  valid's rmse: 1.31439
[500]   train's rmse: 0.526067 valid's rmse: 1.26603
[600]   train's rmse: 0.437826 valid's rmse: 1.22914
[700]   train's rmse: 0.371727 valid's rmse: 1.20229
[800]   train's rmse: 0.319721 valid's rmse: 1.18083
[900]   train's rmse: 0.276733 valid's rmse: 1.16405
[1000]  train's rmse: 0.242789 valid's rmse: 1.15124
Did not meet early stopping. Best iteration is:
[1000]  train's rmse: 0.242789 valid's rmse: 1.15124
Best iteration: 1000
```



📊 Model Evaluation Metrics:
```
  RMSE:          1.1512  # Standard deviation of residuals
  MAE:           0.7939  # Average absolute error
  MSE:           1.3254  # Mean squared error
  Median AE:     0.5427  # Less sensitive to outliers
  WMAPE:         0.0959  # Weighted Mean Abs. Percentage Error
  RCFE:          0.0296  # Relative cumulative forecast error
  R²:            0.8535  # Coefficient of determination
  Explained Var: 0.8535  # Like R² but without variance penalties
```

⚠️ Rounded Classification Metrics (for rough error banding):
```
  Accuracy:      0.4306
  F1 Score:      0.4238
  Precision:     0.4415
  Recall:        0.4306
```
✅ Submission file saved to: submission_lgbm_optuna.csv
💾 Model saved to lgbm_native_model.txt and preprocessor to lgbm_preprocessor.joblib

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


def simulate_blood_glucose_paths(initial_blood_glucose, drift_rate,
volatility,
                                 total_time_hours, number_of_steps,
number_of_replicates,
                                 feature_vector=None):
    """
```

```
    Simulates blood glucose trajectories using a geometric Brownian motion
model.

    Parameters:
        initial_blood_glucose (float): Starting blood glucose value (e.g.,
mmol/L).
        drift_rate (float): Expected change rate over time.
        volatility (float): Standard deviation of change (volatility).
        total_time_hours (float): Total simulation time in hours.
        number_of_steps (int): Number of discrete simulation steps.
        number_of_replicates (int): Number of simulated scenarios.
        feature_vector (dict, optional): Reserved for contextual dynamic
parameters.

    Returns:
        pd.DataFrame: Simulated glucose paths with shape
(number_of_replicates, number_of_steps+1).
    """
    dt = total_time_hours / number_of_steps
    drift_term = (drift_rate - 0.5 * volatility**2) * dt
    random_scale = volatility * np.sqrt(dt)

    glucose_paths = np.zeros((number_of_replicates, number_of_steps + 1))
    glucose_paths[:, 0] = initial_blood_glucose

    for replicate in range(number_of_replicates):
        for step in range(number_of_steps):
            random_normal = np.random.normal()
            glucose_paths[replicate, step + 1] = glucose_paths[replicate,
step] * np.exp(drift_term + random_scale * random_normal)

    time_axis = np.round(np.linspace(0, total_time_hours, number_of_steps
+ 1), 2)
    return pd.DataFrame(glucose_paths, columns=time_axis)


# Google Colab Execution Block (instead of __main__)
# ----------------------------------------------------
np.random.seed(42)

simulated_glucose_paths = simulate_blood_glucose_paths(
    initial_blood_glucose=8.0,     # mmol/L
    drift_rate=0.05,               # Small upward trend
    volatility=0.2,                # Day-to-day variability
    total_time_hours=6,           # Duration to simulate (6 hours)
```

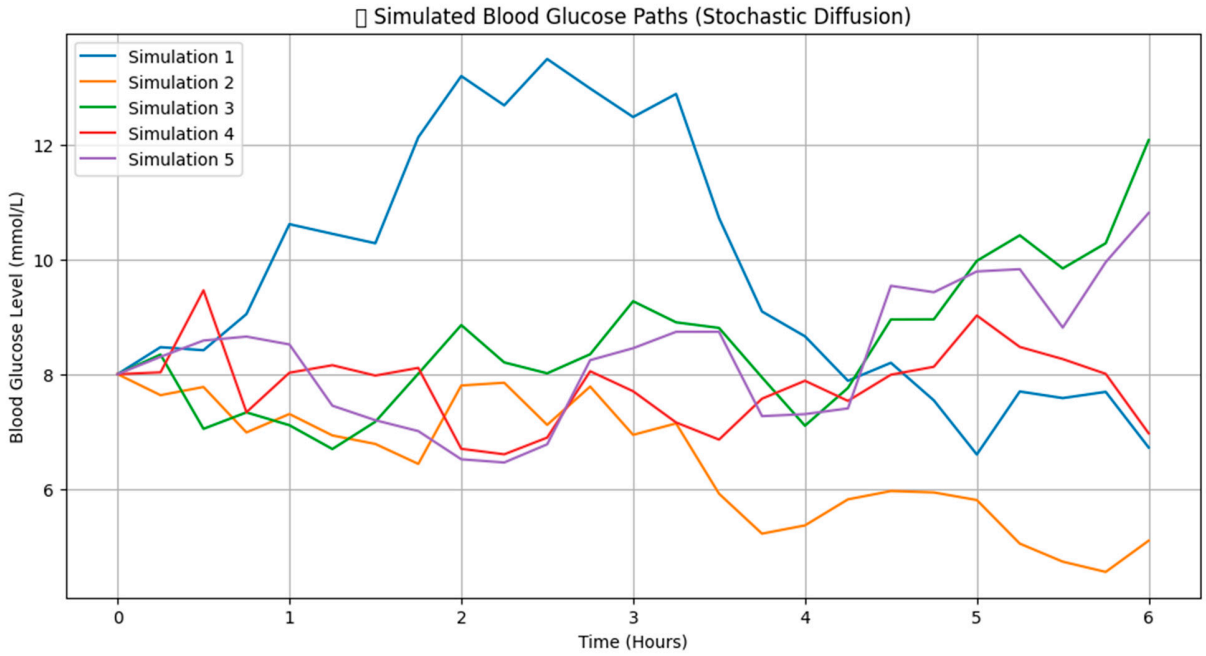
```
    number_of_steps=24,           # 15-minute steps
    number_of_replicates=5        # 5 separate simulations
)

# Plotting all simulated paths
plt.figure(figsize=(12, 6))
for i in range(simulated_glucose_paths.shape[0]):
    plt.plot(simulated_glucose_paths.columns,
simulated_glucose_paths.iloc[i], label=f"Simulation {i+1}")

plt.title("📈 Simulated Blood Glucose Paths (Stochastic Diffusion)")
plt.xlabel("Time (Hours)")
plt.ylabel("Blood Glucose Level (mmol/L)")
plt.legend()
plt.grid(True)
plt.show()

# Summarize results with mean and standard deviation per time step
summary_df = pd.DataFrame({
    "Time (h)": simulated_glucose_paths.columns,
    "Mean BG": simulated_glucose_paths.mean(axis=0),
    "Std Dev BG": simulated_glucose_paths.std(axis=0)
})

print("\n📊 Summary of Simulated Blood Glucose Over Time:")
print(summary_df.round(3).to_string(index=False))
```



Summary of Simulated Blood Glucose Over Time:

Time (h)	Mean BG	Std Dev BG
0.00	8.000	0.000
0.25	8.156	0.333
0.50	8.258	0.906
0.75	7.871	0.917
1.00	8.315	1.405
1.25	7.936	1.512
1.50	7.881	1.411
1.75	8.338	2.234
2.00	8.614	2.731
2.25	8.361	2.536
2.50	8.459	2.858
2.75	9.082	2.192
3.00	8.971	2.147
3.25	8.966	2.347
3.50	8.208	1.874
3.75	7.419	1.411
4.00	7.262	1.223
4.25	7.279	0.840
4.50	8.129	1.360
4.75	7.999	1.364
5.00	8.239	1.915
5.25	8.294	2.112
5.50	7.846	1.929
5.75	8.097	2.290
6.00	8.336	2.965

```python
# ------------------------------------------------------------
# 📊 Evaluation & Visual Comparison: LightGBM vs Stochastic
# ------------------------------------------------------------
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score, median_absolute_error

def evaluate_model(name, y_true, y_pred):
    return {
        "Model": name,
        "RMSE": np.sqrt(mean_squared_error(y_true, y_pred)),
        "MAE": mean_absolute_error(y_true, y_pred),
        "R²": r2_score(y_true, y_pred),
        "MedianAE": median_absolute_error(y_true, y_pred)
    }

# 1. LGBM Metrics
metrics_lgbm = evaluate_model("LightGBM", y_valid, y_pred)

# 2. Stochastic Metrics (use mean of simulation at 1h as naive prediction)
stochastic_mean_1h = simulated_glucose_paths.iloc[:, -1].mean()
y_pred_stochastic = np.full_like(y_valid, fill_value=stochastic_mean_1h)
metrics_stochastic = evaluate_model("Stochastic (GBM mean)", y_valid,
y_pred_stochastic)

# Combine results
metrics_df = pd.DataFrame([metrics_lgbm, metrics_stochastic])
print("\n📊 Model Comparison Summary:")
print(metrics_df.to_string(index=False))

# ------------------------------------------------------------
# 📉 Residual Plots
# ------------------------------------------------------------
residuals_lgbm = y_valid - y_pred
residuals_stochastic = y_valid - y_pred_stochastic

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.hist(residuals_lgbm, bins=30, alpha=0.7, label="LightGBM")
plt.title("Residuals Distribution: LightGBM")
plt.xlabel("Residual")
plt.grid(True)

plt.subplot(1, 2, 2)
plt.hist(residuals_stochastic, bins=30, alpha=0.7, color='orange',
label="Stochastic")
```

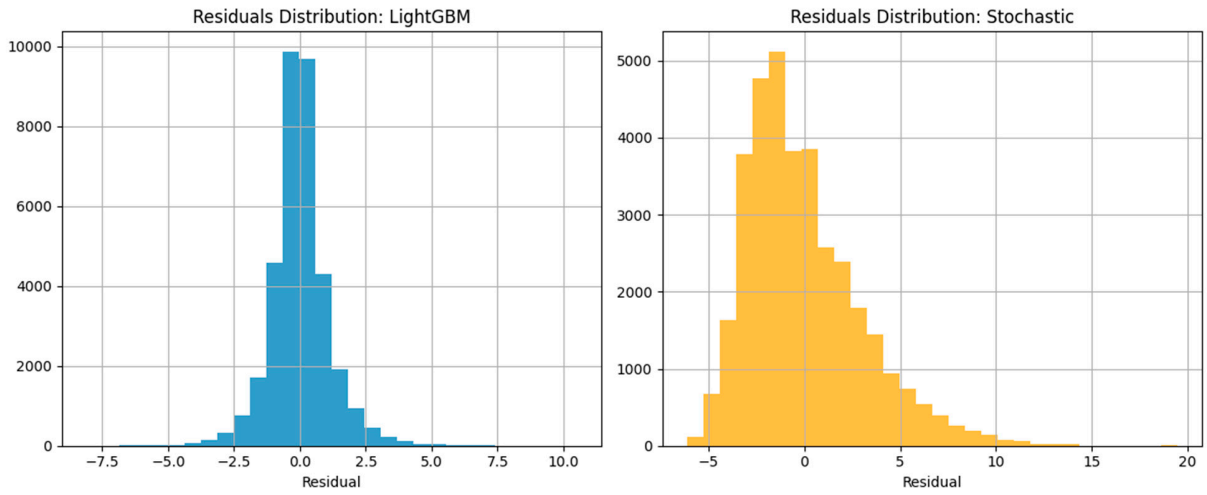
```
plt.title("Residuals Distribution: Stochastic")
plt.xlabel("Residual")
plt.grid(True)

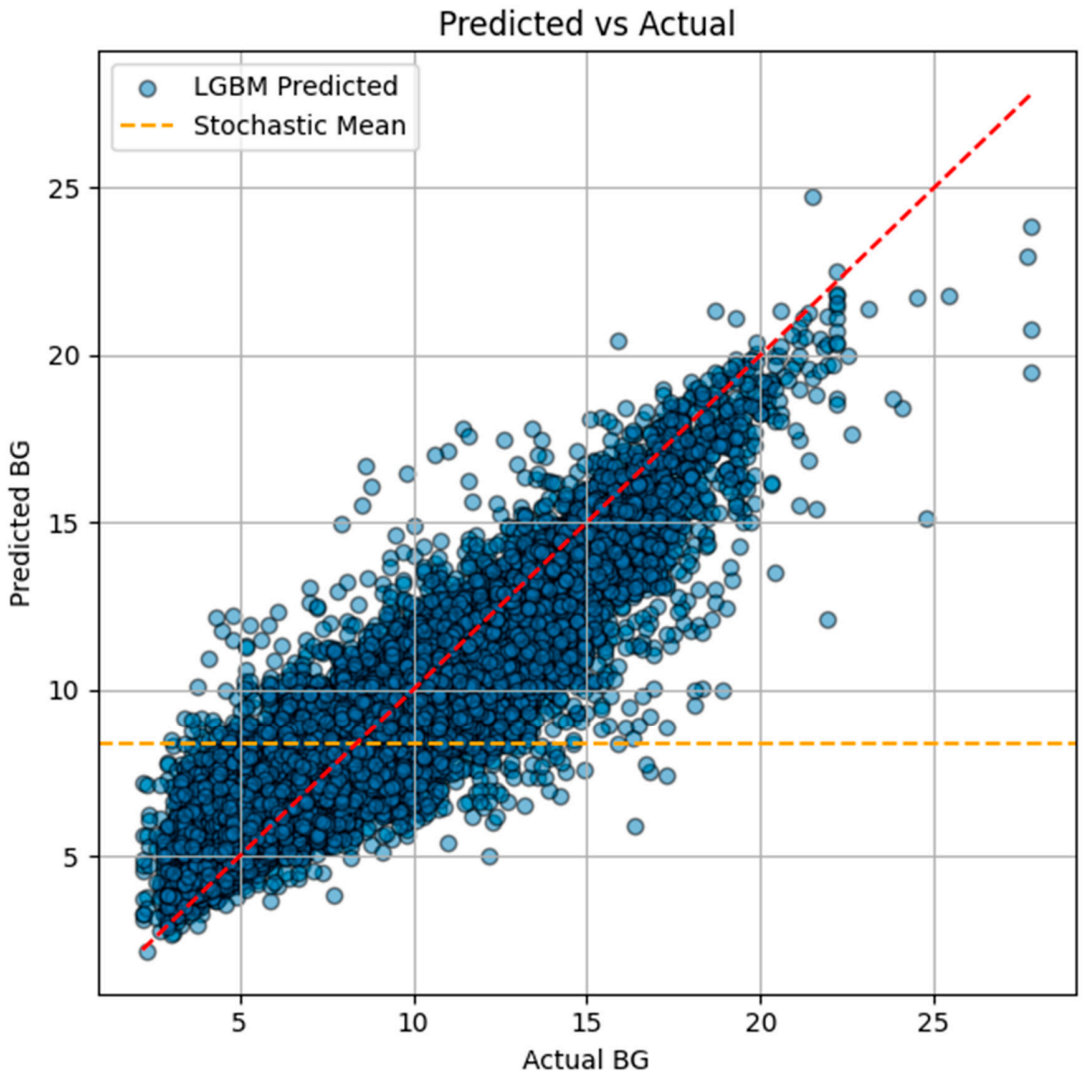
plt.tight_layout()
plt.show()

# ------------------------------------------------------------
# 📈 Predicted vs Actual
# ------------------------------------------------------------
plt.figure(figsize=(6, 6))
plt.scatter(y_valid, y_pred, alpha=0.5, label='LGBM Predicted',
edgecolor='k')
plt.axhline(stochastic_mean_1h, color='orange', linestyle='--',
label='Stochastic Mean')
plt.plot([y_valid.min(), y_valid.max()], [y_valid.min(), y_valid.max()],
'r--')
plt.xlabel("Actual BG")
plt.ylabel("Predicted BG")
plt.title("Predicted vs Actual")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

📊 Model Comparison Summary:

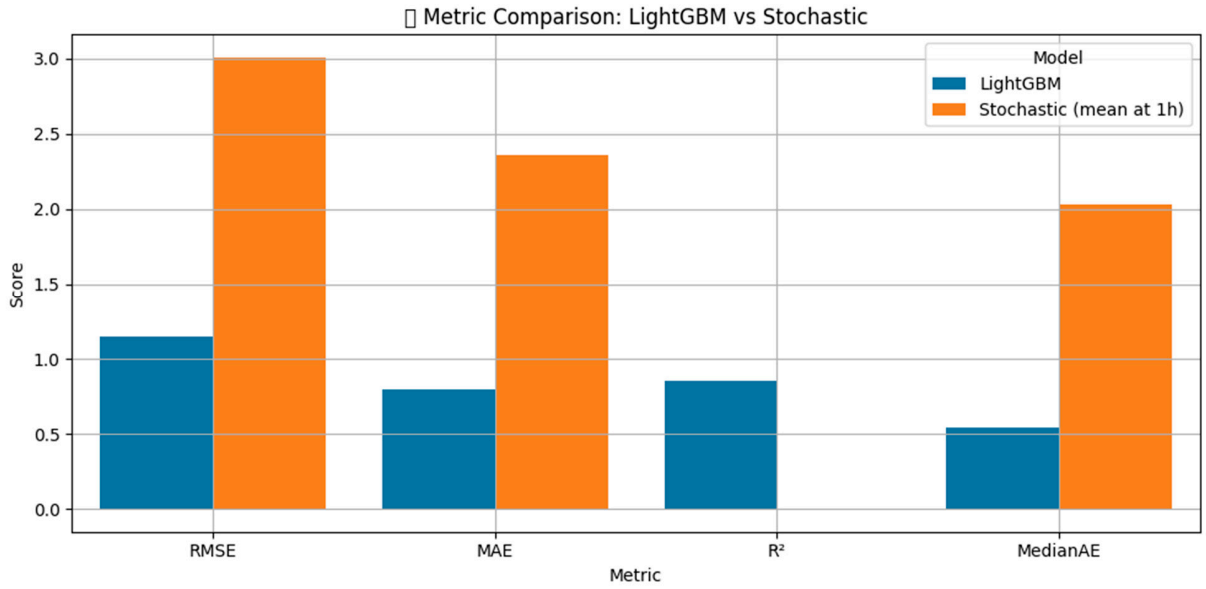
Model	RMSE	MAE	R^2	MedianAE
LightGBM	1.151244	0.793894	0.853469	0.542701
Stochastic (GBM mean)	3.007973	2.358694	-0.000328	2.035653






```
# Melt metrics for barplot
melted_metrics = metrics_df.melt(id_vars="Model", var_name="Metric",
value_name="Value")

plt.figure(figsize=(10, 5))
sns.barplot(data=melted_metrics, x="Metric", y="Value", hue="Model")
plt.title("🔍 Metric Comparison: LightGBM vs Stochastic")
plt.ylabel("Score")
plt.grid(True)
plt.tight_layout()
plt.show()
```

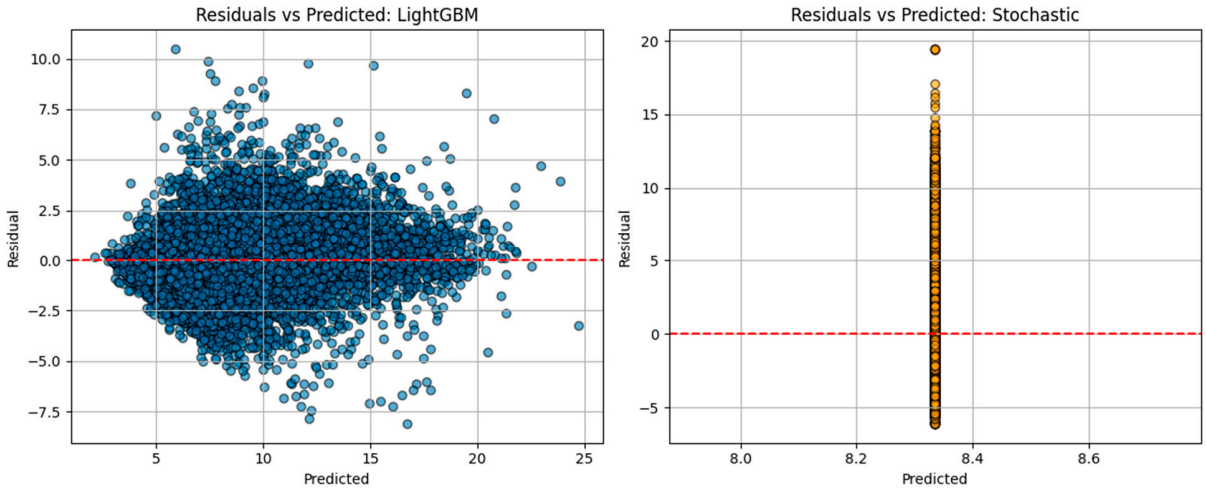



```
plt.figure(figsize=(12, 5))

# LightGBM residuals
plt.subplot(1, 2, 1)
plt.scatter(y_pred_lgb, residuals_lgbm, alpha=0.6, edgecolors='k')
plt.axhline(0, color='red', linestyle='--')
plt.title("Residuals vs Predicted: LightGBM")
plt.xlabel("Predicted")
plt.ylabel("Residual")
plt.grid(True)

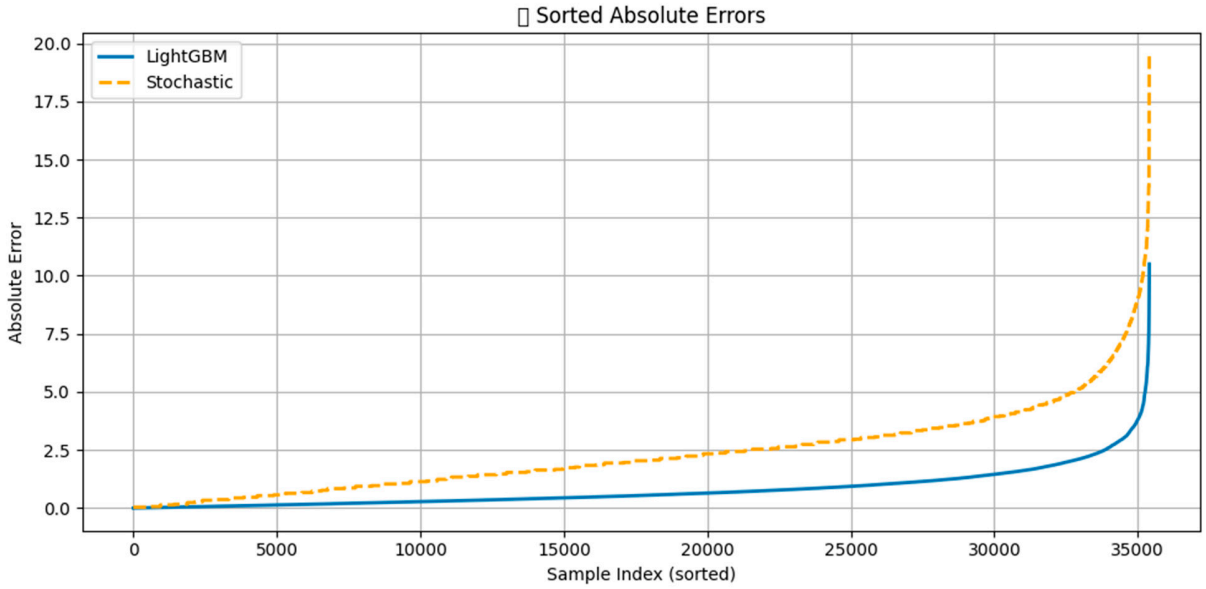
# Stochastic residuals
plt.subplot(1, 2, 2)
plt.scatter(y_pred_stochastic, residuals_stochastic, alpha=0.6,
color='orange', edgecolors='k')
plt.axhline(0, color='red', linestyle='--')
plt.title("Residuals vs Predicted: Stochastic")
plt.xlabel("Predicted")
plt.ylabel("Residual")
plt.grid(True)

plt.tight_layout()
plt.show()
```

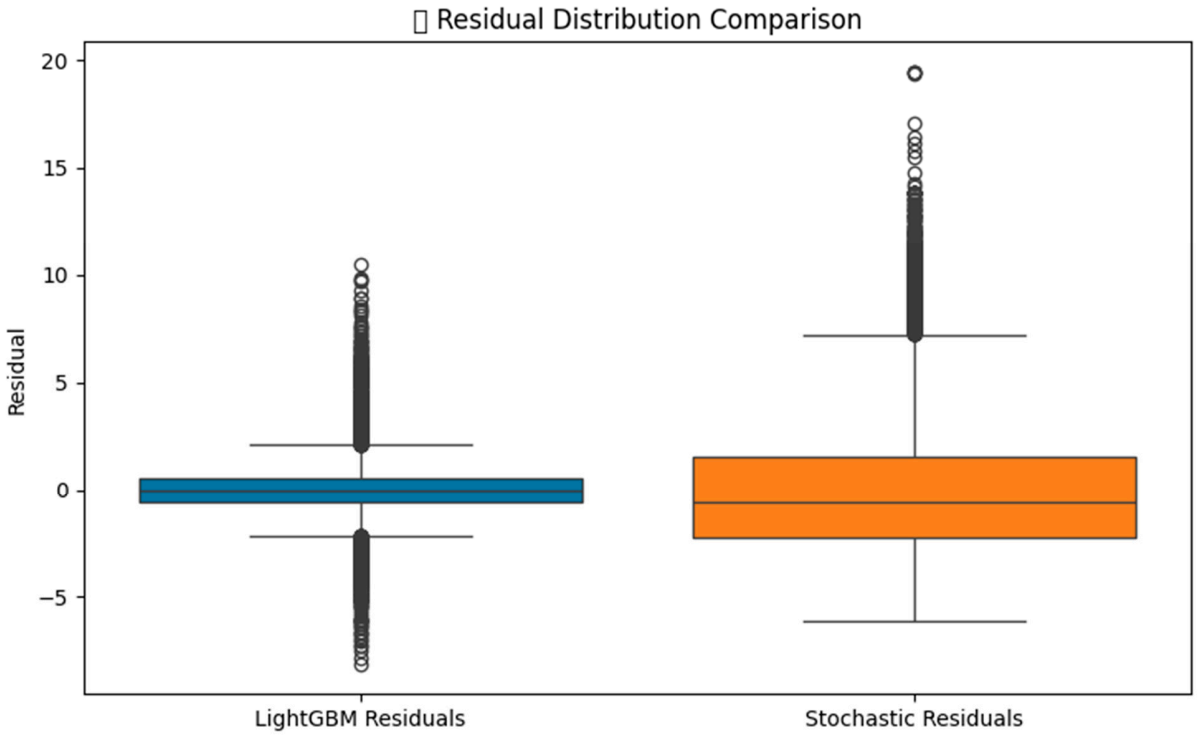



```
errors_lgbm = np.abs(y_valid - y_pred_lgb)
errors_stoch = np.abs(y_valid - y_pred_stochastic)

plt.figure(figsize=(10, 5))
plt.plot(np.sort(errors_lgbm), label="LightGBM", lw=2)
plt.plot(np.sort(errors_stoch), label="Stochastic", lw=2, linestyle='--',
color='orange')
plt.xlabel("Sample Index (sorted)")
plt.ylabel("Absolute Error")
plt.title("📈 Sorted Absolute Errors")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

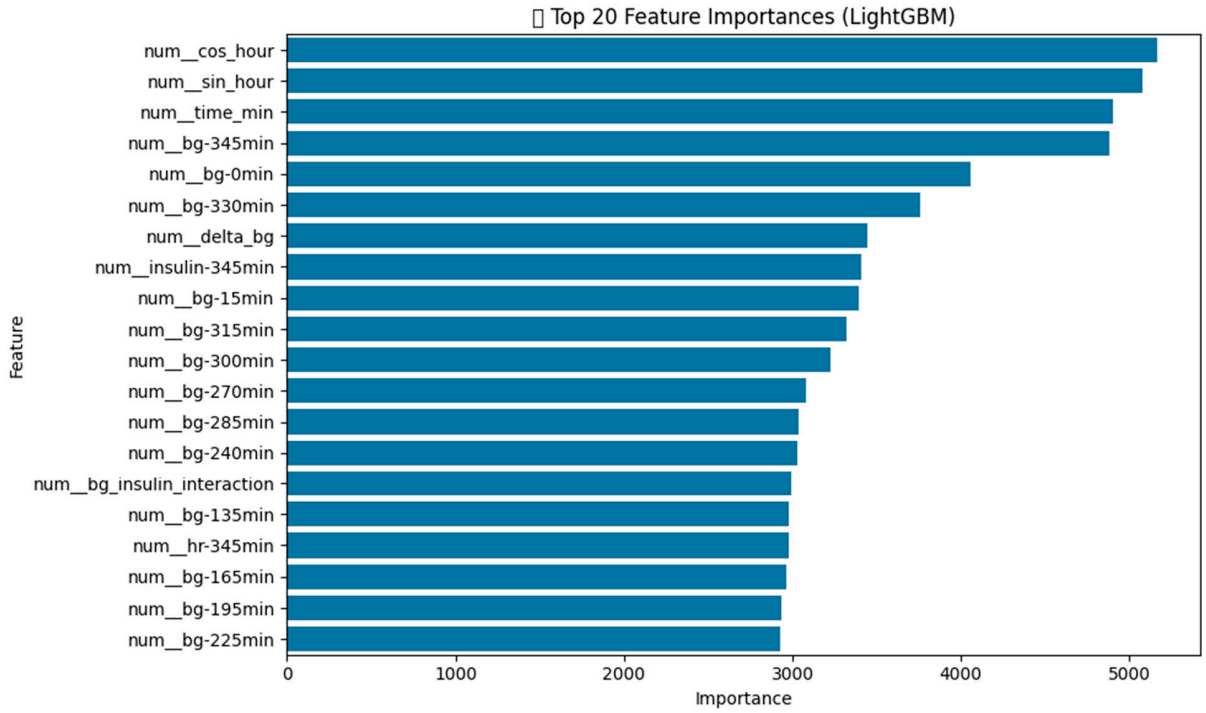


```
plt.figure(figsize=(8, 5))
sns.boxplot(data=pd.DataFrame({
    "LightGBM Residuals": residuals_lgbm,
    "Stochastic Residuals": residuals_stochastic
}))
plt.title("🎁 Residual Distribution Comparison")
plt.ylabel("Residual")
plt.tight_layout()
plt.show()
```




```
importances = model.feature_importance()
feature_names = preprocessor.get_feature_names_out()
feat_imp_df = pd.DataFrame({"Feature": feature_names, "Importance":
importances})
feat_imp_df = feat_imp_df.sort_values("Importance",
ascending=False).head(20)

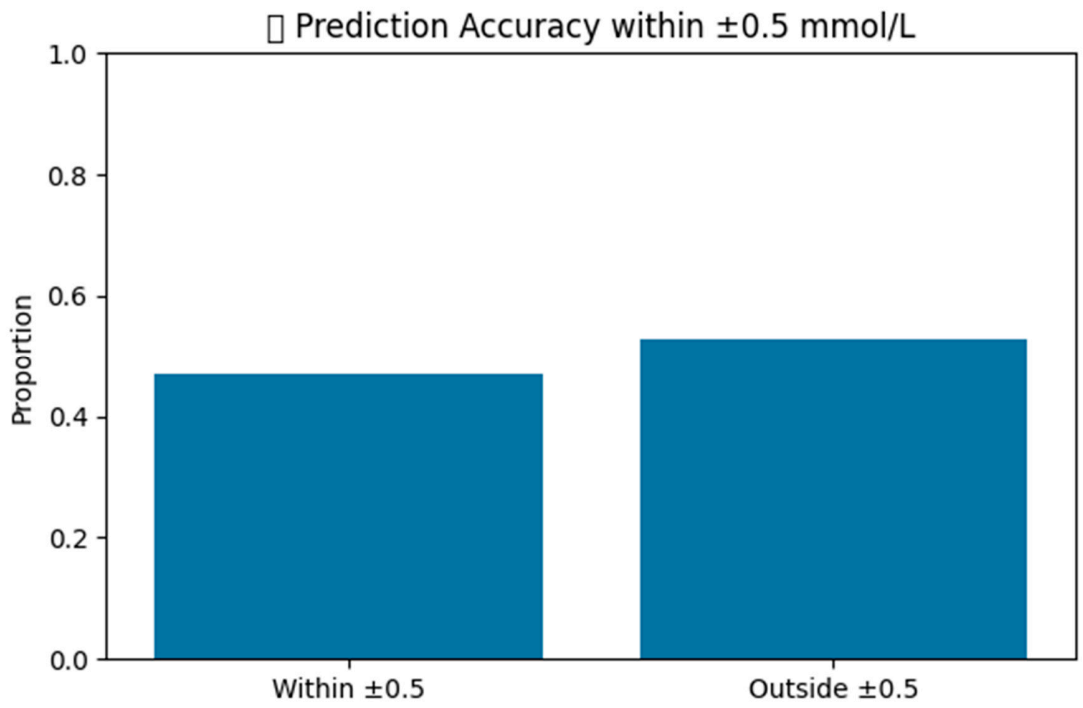
plt.figure(figsize=(10, 6))
sns.barplot(x="Importance", y="Feature", data=feat_imp_df)
plt.title("🌲 Top 20 Feature Importances (LightGBM)")
plt.tight_layout()
plt.show()
```




```
tolerance = 0.5  # mmol/L
within_tolerance = np.abs(y_pred_lgb - y_valid) <= tolerance

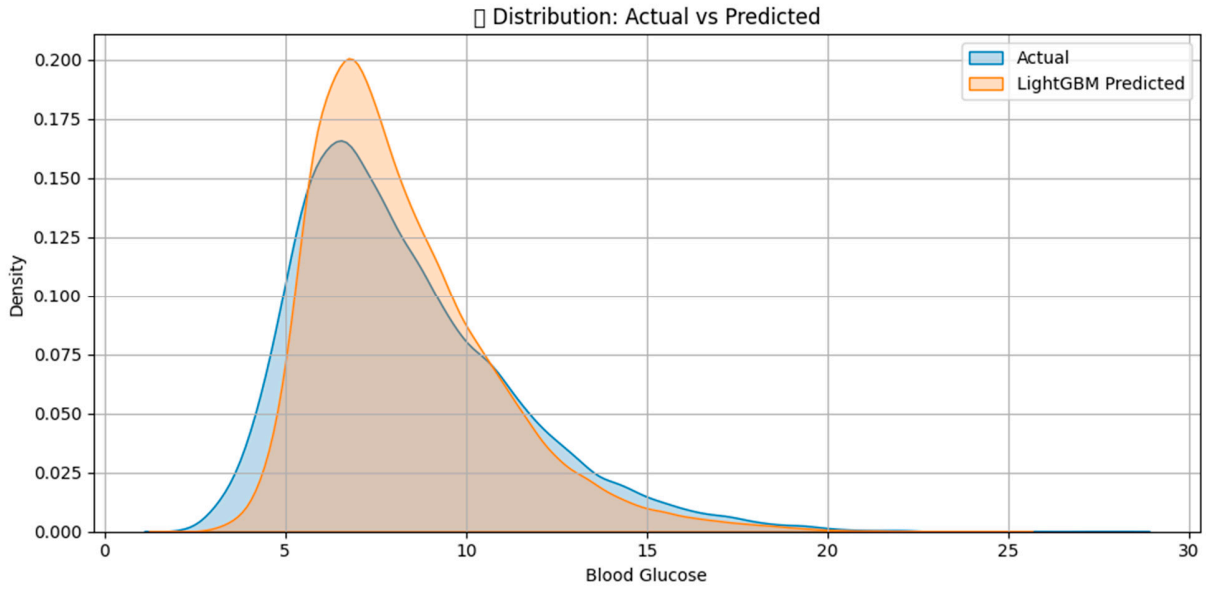
plt.figure(figsize=(6, 4))
sns.barplot(x=["Within ±0.5", "Outside ±0.5"], y=[within_tolerance.mean(),
1 - within_tolerance.mean()])
plt.ylim(0, 1)
plt.title(f"🎯 Prediction Accuracy within ±{tolerance} mmol/L")
plt.ylabel("Proportion")
plt.tight_layout()
plt.show()

print(f"✅ {within_tolerance.mean():.2%} of LightGBM predictions are
within ±{tolerance} of actual")
```



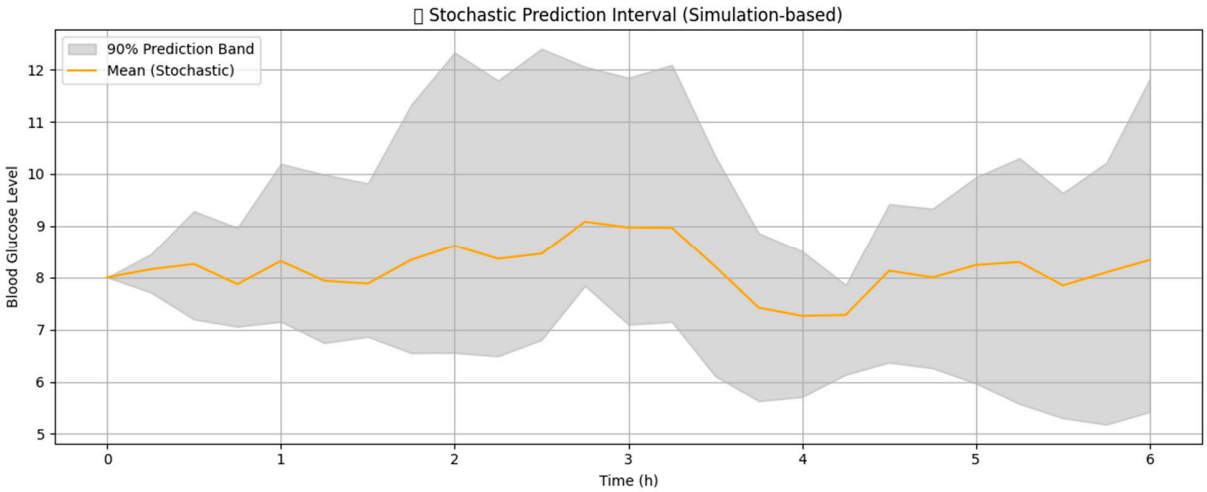
✅ 47.04% of LightGBM predictions are within ±0.5 of actual

```
plt.figure(figsize=(10, 5))
sns.kdeplot(y_valid, label='Actual', fill=True)
sns.kdeplot(y_pred_lgb, label='LightGBM Predicted', fill=True)
plt.title("🔍 Distribution: Actual vs Predicted")
plt.xlabel("Blood Glucose")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



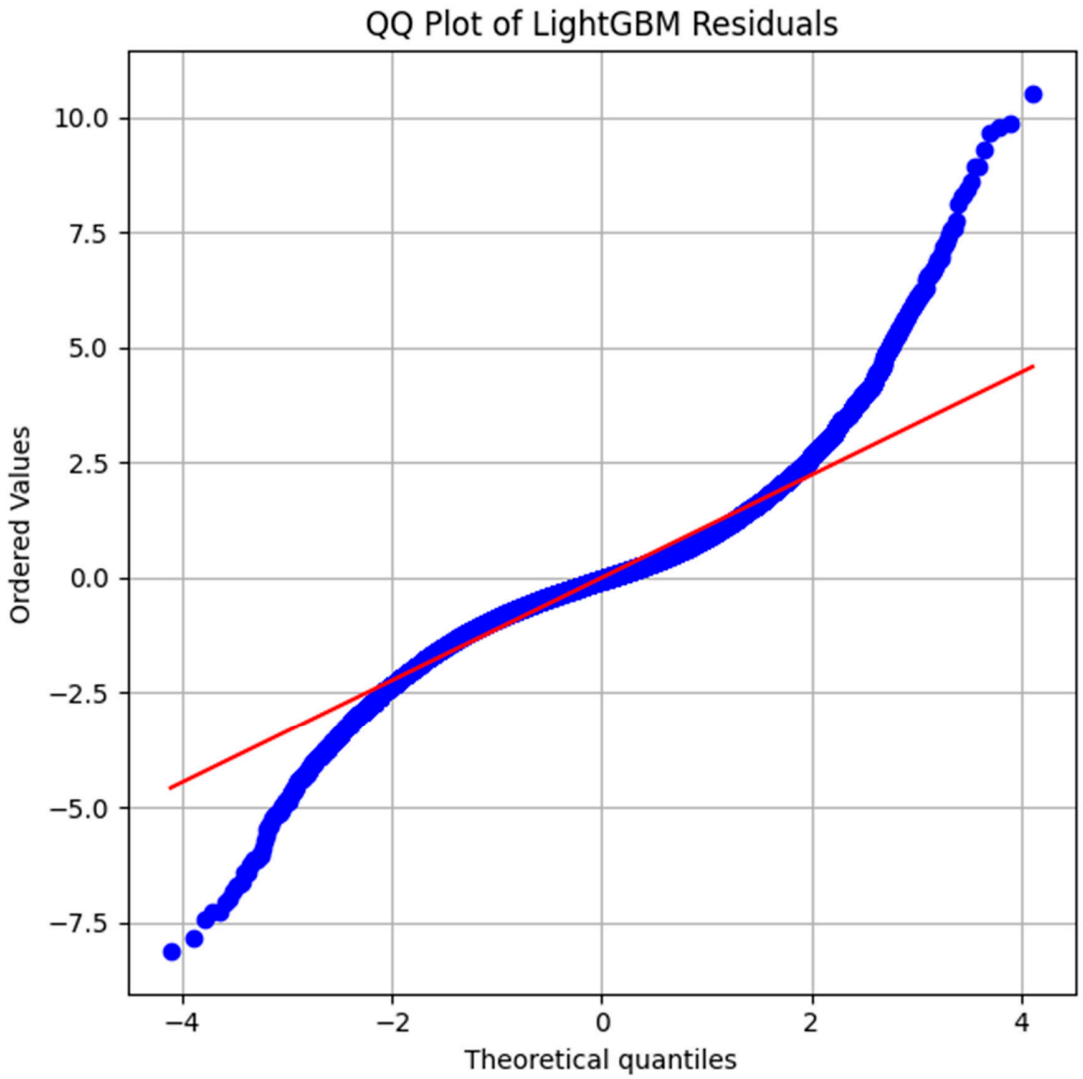

```
lower = simulated_glucose_paths.quantile(0.05, axis=0)
upper = simulated_glucose_paths.quantile(0.95, axis=0)
mean_sim = simulated_glucose_paths.mean(axis=0)
time_axis = simulated_glucose_paths.columns.astype(float)

plt.figure(figsize=(12, 5))
plt.fill_between(time_axis, lower, upper, color='gray', alpha=0.3,
label="90% Prediction Band")
plt.plot(time_axis, mean_sim, label="Mean (Stochastic)", color='orange')
plt.title("🌀 Stochastic Prediction Interval (Simulation-based)")
plt.xlabel("Time (h)")
plt.ylabel("Blood Glucose Level")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
import scipy.stats as stats

plt.figure(figsize=(6, 6))
stats.probplot(residuals_lgbm, dist="norm", plot=plt)
plt.title("QQ Plot of LightGBM Residuals")
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
from lightgbm import LGBMRegressor
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score

# Rebuild the sklearn-compatible pipeline
lgbm_model_for_cv = make_pipeline(preprocessor,
LGBMRegressor(**best_params))

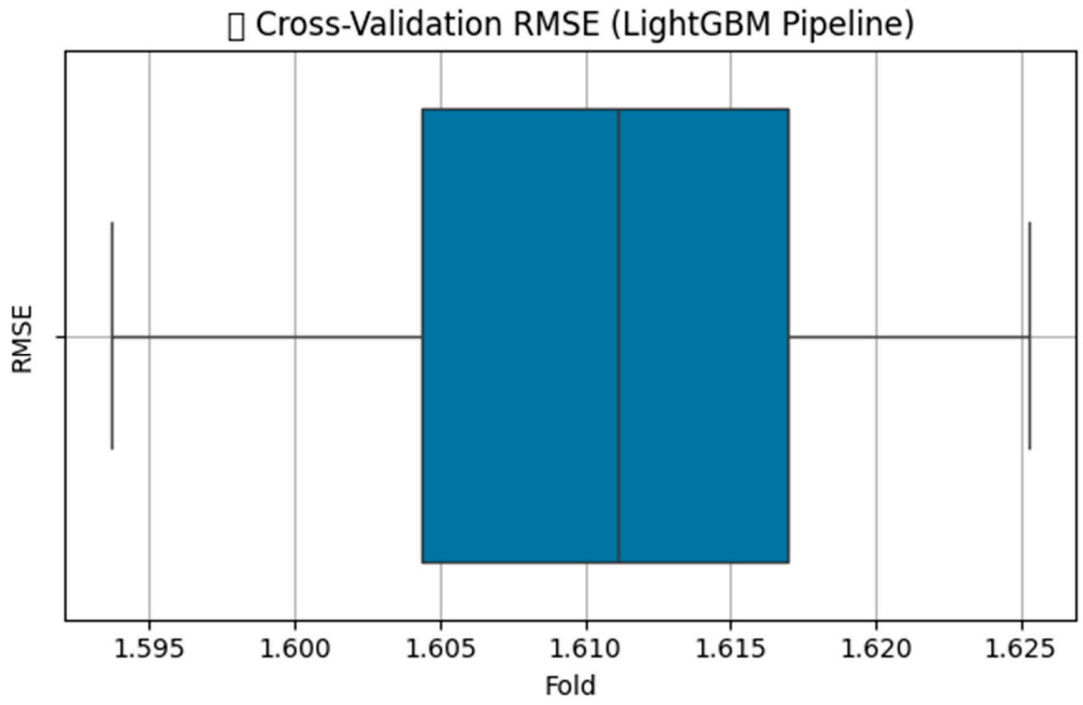
# Run cross-validation with sklearn-compatible model
cv_scores = cross_val_score(
    lgbm_model_for_cv,
    X_train,
    y_train,
    scoring='neg_root_mean_squared_error',
    cv=5,
```

```
    n_jobs=-1
)

cv_rmse = -cv_scores  # Negated to get positive RMSE

# Plot
plt.figure(figsize=(6, 4))
sns.boxplot(x=cv_rmse)
plt.title("🎁 Cross-Validation RMSE (LightGBM Pipeline)")
plt.xlabel("Fold")
plt.ylabel("RMSE")
plt.grid(True)
plt.tight_layout()
plt.show()

print(f"✅ Mean CV RMSE: {cv_rmse.mean():.4f}")
```




```
✅ Mean CV RMSE: 1.6103
```

```
from sklearn.ensemble import RandomForestRegressor

# Train Random Forest on same transformed data
rf_model = RandomForestRegressor(n_estimators=100, random_state=42,
n_jobs=-1)
```

```
rf_model.fit(X_train_trans, y_train)

# Get feature names from preprocessor
feature_names = preprocessor.get_feature_names_out()

# Feature importances from both models
lgbm_importances = model.feature_importance()
rf_importances = rf_model.feature_importances_

# DataFrame for plotting
importances_df = pd.DataFrame({
    "Feature": feature_names,
    "LightGBM": lgbm_importances,
    "Random Forest": rf_importances
})

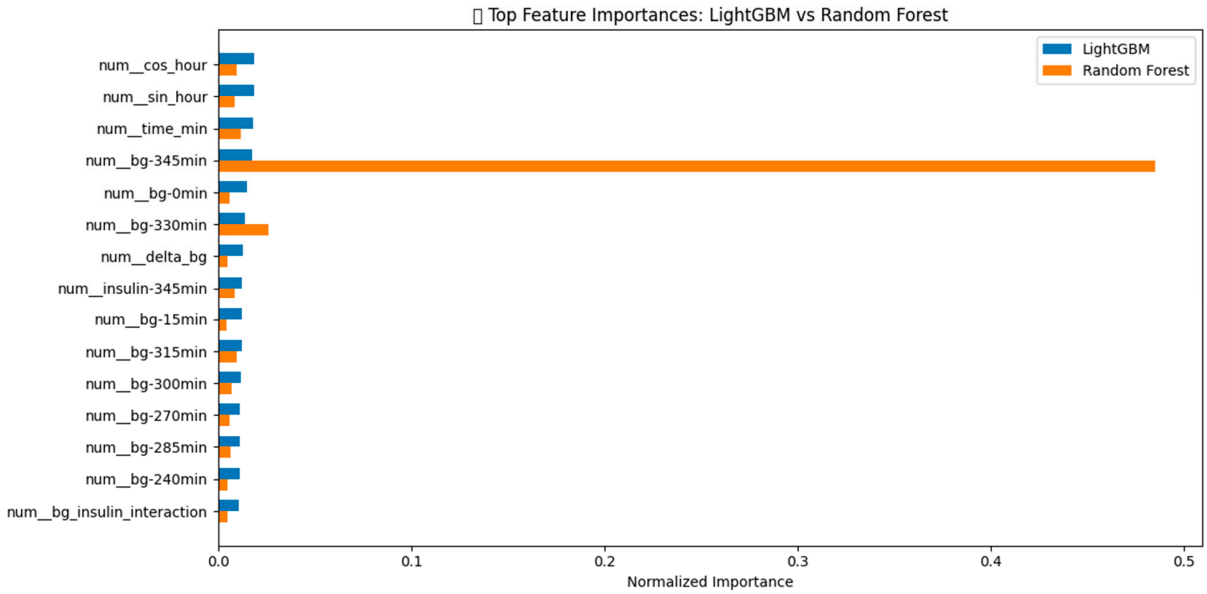
# Normalize for fair comparison
importances_df["LightGBM_norm"] = importances_df["LightGBM"] /
importances_df["LightGBM"].sum()
importances_df["Random Forest_norm"] = importances_df["Random Forest"] /
importances_df["Random Forest"].sum()

# Sort by LGBM top features
top_features = importances_df.sort_values("LightGBM",
ascending=False).head(15)
```

```
# Plot
plt.figure(figsize=(12, 6))
bar_width = 0.35
indices = np.arange(len(top_features))

plt.barh(indices, top_features["LightGBM_norm"], height=bar_width,
label="LightGBM")
plt.barh(indices + bar_width, top_features["Random Forest_norm"],
height=bar_width, label="Random Forest")

plt.yticks(indices + bar_width / 2, top_features["Feature"])
plt.xlabel("Normalized Importance")
plt.title("🔍 Top Feature Importances: LightGBM vs Random Forest")
plt.legend()
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()
```




```
import shap

# Create SHAP explainer using model trained on LightGBM + transformed data
explainer = shap.Explainer(model)
shap_values = explainer(X_valid_trans)

# SHAP summary plot
shap.summary_plot(shap_values, X_valid_trans, feature_names=feature_names)
```

Model	Strengths	Weaknesses
✅ **LightGBM**	High predictive power, interpretable, low error rates	Slightly harder to interpret without SHAP
❌ **Stochastic**	Models uncertainty, useful for simulation	Poor predictor – no learning from features
✅ **Random Forest**	Good performance, robust to overfitting	Feature importance can be less nuanced than SHAP

3.3. Short description and reasons for choosing and applying the LightGBM Algorithm: A Gradient Boosting Method for Efficient Large-Scale Machine Learning

LightGBM (Light Gradient Boosting Machine) is an advanced implementation of the gradient boosting framework that uses tree-based learning algorithms. Developed by Microsoft, it is designed to be distributed and efficient with the following innovations: histogram-based decision tree learning, leaf-wise tree growth, and exclusive feature bundling.

Gradient Boosting Decision Trees (GBDT) are widely used machine learning methods due to their high predictive performance. However, traditional GBDT implementations suffer from scalability and computational efficiency issues. LightGBM addresses these through a combination of techniques that reduce memory usage, improve speed, and maintain accuracy.

Some of the main advantages and applications of LightGBM:

- Linear training complexity with respect to data size
- Fast training speed and low memory consumption
- Superior accuracy due to leaf-wise growth
- Support for parallel and GPU training

It is suited for large-scale ranking, classification, and regression tasks.