ipt Instituto Politécnico de Tomar

Polytechnic University of Tomar

IoT Driven Machine Learning Predictive Models Applied to Irrigation Optimization in Tomato Cultivation

Master's Thesis

Maung Maung Htwe

Master's in Analytics and Organizational Intelligence

Tomar/ November/ 2025

Polytechnic University of Tomar

IoT Driven Machine Learning Predictive Models Applied to Irrigation Optimization in Tomato Cultivation

Master's Thesis

Maung Maung Htwe

Supervised by:
Professor Doctor Sandra Maria Gonçalves Vilas Boas Jardim – Polytechnic University of Tomar

Jury
Coordinating Professor Sandra Maria Gonçalves de Vilas Boas Jardim
Adjunct Professor Rolando Lúcio Germano Miragaia (Arguente)
Adjunct Professor João Manuel Mourão Patrício

Thesis presented to the Polytechnic University of Tomar to fulfill the requirements necessary for obtaining the Master's degree in Analytics and Organizational Intelligence

Dreaming is human, measuring is scientific,
transformaing data to knowledge is
the purpose of a master's degree.

ACKNOWLEDGEMENTS

Completing a study of this nature represents not only the closure of an academic stage but also the consolidation of a personal and professional journey marked by learning, challenges, and overcoming obstacles.

I first thank the faculty and supervisors who accompanied me throughout this work for their scientific support, critical guidance, and constant encouragement. Their contributions were essential for the realization of this research. In particular, during my Erasmus+ mobility, I extend special gratitude to my hosting institution supervisors at the Climate, Atmosphere and Water Research Institute – Bulgarian Academy of Sciences (CAWRI-BAS), Prof. Lachezar Filchev and Prof. Ekaterina Batchvarova, for generously providing access to their facilities, resources, and expertise. Equally, I am deeply thankful to my sending institution's program coordinator and supervisor at the Smart Cities Research Center, Polytechnic University of Tomar, Prof. Sandra Jardim, for her unwavering mentorship, strategic oversight, and facilitation of this international opportunity.

I also express sincere appreciation to Martina Galaverni, Giulia Oddi., et al. for making their comprehensive "IoT-based Data Collection in a Tomato Cultivation Under Different Irrigation Regimes" dataset publicly available through Mendeley Data. This reliable, open-access resource was foundational to my analysis, enabling rigorous model development and validation without which this study would not have been possible.

Finally, I leave a word of gratitude to my family and friends for the emotional support, motivation, and understanding demonstrated throughout this entire process. Without their encouragement, this journey would not have been possible.

To all, my deepest thanks.

RESUMO

O presente estudo centrou-se no desenvolvimento de um framework de análise preditiva impulsionado por Internet das Coisas (IoT) para a otimização dinâmica de irrigação no cultivo de tomate, abordando os desafios críticos de segurança alimentar global e escassez de água. Utilizando um conjunto de dados experimental abrangente, controlado e multi-sensor, coletado em Parma, Itália, de 29 de junho a 13 de setembro de 2023, o framework integra dados ambientais (temperatura do ar, humidade, CO_2, pressão) e parâmetros de solo chave (humidade, temperatura, condutividade elétrica). Através de uma pipeline de pré-processamento rigorosa e engenharia de features adaptada, foram extraídos padrões temporais, relações inter-variáveis e indicadores agronómicos como os Growing Degree Days (GDD) e Soil Moisture Deficit (SMD).

Desenvolveu-se um modelo de regressão eXtreme Gradient Boosting (XGBoost) de duas partes (classificação + regressão) para prever precisamente o volume diário de água necessário por hectare, alcançando um RMSE de 6.37 m^3/ha/dia e volumétrico de R^2 de 0.95 no conjunto de teste.

A inovação reside na capacidade de aproveitar dados históricos IoT complexos para construir uma camada de inteligência para agendamento de irrigação. A precisão do modelo em identificar níveis ótimos de água sob condições variáveis é demonstrada numa simulação de otimização dinâmica usando o algoritmo Sequential Least Squares Programming (SLSQP), alcançando poupanças significativas de água de 50.84% vs. predições raw e 43.97% vs. baseline constante. Esta pesquisa fornece insights data-driven fundamentais para estratégias de irrigação de precisão altamente eficazes, empoderando agricultores a reduzir desperdício de água e prevenir sobre-irrigação prejudicial, levando a uma agricultura inteligente mais sustentável e eficiente.

Palavras-chave: Internet das Coisas (IoT); Análise Preditiva; Otimização de Irrigação; Gestão de Água; Saúde do Solo; Machine Learning; Modelo XGBoost.

ABSTRACT

This study focuses on developing an Internet of Things (IoT)-driven predictive analytics framework for dynamic irrigation optimization in tomato cultivation, addressing critical challenges of global food security and water scarcity. Leveraging a comprehensive, experimentally controlled multi-sensor dataset collected in Parma, Italy, from June 29 to September 13, 2023, the framework integrates environmental data (air temperature, humidity, CO_2, pressure) and key soil parameters (humidity, temperature, electrical conductivity). Through a rigorous data preprocessing pipeline and tailored feature engineering, temporal patterns, inter-variable relationships, and agronomic indicators like Growing Degree Days (GDD) and Soil Moisture Deficit (SMD) were extracted.

A two-part eXtreme Gradient Boosting (XGBoost) regression model (combining classification and regression) was developed and validated to precisely predict daily water volume per hectare, achieving an RMSE of 6.37 m^3/ha/day and a volumetric R^2 of 0.95 on the test set.

The innovation lies in harnessing complex historical IoT data to build an intelligence layer for irrigation scheduling. The model's accuracy in identifying optimal water levels under varying conditions is demonstrated in a dynamic optimization simulation using the Sequential Least Squares Programming (SLSQP) algorithm, achieving significant water savings of 50.84% vs. raw predictions and 43.97% vs. constant baseline. This research provides foundational data-driven insights for highly effective precision irrigation strategies, empowering farmers to reduce water waste and prevent harmful over-irrigation, leading to more sustainable and efficient smart agriculture.

Keywords: Internet of Things (IoT); Predictive Analytics; Irrigation Optimization; Water Management; Soil Health; Machine Learning; XGBoost Model.

TABLE OF CONTENTS

Acknowledgements I
Resumo II
Abstract III
Table of Contents IV
List of Figures VI
List of Tables VII
List of Abbreviations VIII
1. Introduction and Literature Review 9
  1.1. National and Global Framework of Water Resource Management in Agriculture 9
  1.2. Driving Objectives of the Research 11
  1.3. Review of Predictive Models and Optimization in SA 12
  1.4. Thesis Structure 12
2. Materials and Methods 14
  2.1. Experimental Data Acquisition and Description 14
  2.2. Data Preprocessing and Synchronization 15
  2.3. Feature Engineering and Agronomic Intelligence 17
3. Predictive Modeling and Comparative 20
  3.1. Time-Series Data Splitting and Scaling 20
  3.2. Two-Part XGBoost Model Implementation 20
  3.3. Comparative Benchmarking and Model Selection 21
4. Dynamic Optimization Framework 22
  4.1. Theoretical Foundations of SLSQP Optimization Algorithm 22
  4.2. Objective Function and Agronomic Constraints 22
5. Results, Interpretation and Discussion 24
  5.1. Model Performance and Interpretation on XGBoost 24
  5.2. Feature Importance and Sensitivity Analysis 25
  5.3. Dynamic Optimization Results and Water Savings 26
  5.4. Comparative Novelty Assessment and Practical Implications 28
6. Conclusions and Future Work 30
  6.1. Summary of Findings and Contributions 30
  6.2. Limitations and Future Research Directions 30
    6.2.1. Limitations of the Current Study 30
    6.2.2. Future Research Directions 31
Bibliographic References 32

Appendices..37
I. Source Code 37
II. Description of Datasets 70

LIST OF FIGURES

Figure 1. Global agricultural water consumption....10
Figure 2. Smart irrigation systems using advanced technologies....11
Figure 3. Flowchart of the IoT-driven predictive analytics and dynamic optimization framework....14
Figure 4. Raw IoT sensor data (a) Soil moisture content per irrigation line over time (b) Cumulative water volume per irrigation line over time (m^3)....16
Figure 5. Unified daily water and soil parameters (a) Daily mean soil moisture content over time (%) (b) Total daily water volume over time (m^3/day)....16
Figure 6. Processed environmental and agronomic parameters (a) 7-day rolling mean soil moisture content (%) (b) Calculated daily growing degree days (GDD) (°C-day)....19
Figure 7. Correlation heatmap of selected features and target variable....19
Figure 8. Two-part XGBoost model prediction performance for daily water volume (a) Comparison of actual and predicted daily water volume on the test set (b) Comparison of actual and predicted daily water volume on the training set....25
Figure 9. XGBoost regressor feature importance (top 15 by gain)....25
Figure 10. Dynamic irrigation optimization results (a) Daily water volume comparison: actual, model predicted, and optimized applied over test period (m^3/ha/day) (b) Soil moisture trajectory and optimized irrigation events over test period (%)....28

LIST OF TABLES

Table 1. Experimental soil, water and environmental summary statistics (Aggregated daily means)……..…………………………………………………………………………..........15
Table 2. Optimal XGBoost hyperparameters..20
Table 3. Comparative analysis of two-part predictive model performance on test set..................21
Table 4. Overall two-part XGBoost model performance on test set...21
Table 5. Dynamic optimization simulation results (test period)...27
Table 6. Comparative analysis of ML models for irrigation prediction 28

LIST OF ABBREVIATIONS

BI – Business Intelligence

CSV – Comma-Separated Values

DT – Digital Twin

EC – Electrical Conductivity

ETc – Crop Evapotranspiration

FAO – Food and Agriculture Organization

GDD – Growing Degree Days

IoT – Internet of Things

KPI – Key Performance Indicator

LSTM-RNN – Long Short Term Memory - Recurrent Neural Network

MAE – Mean Absolute Error

ML – Machine Learning

RMSE – Root Mean Squared Error

R^2 – R-squared

SA – Smart Agriculture

SMD – Soil Moisture Deficit

SVM – Support Vector Machines

SVR – Support Vector Regression

XGBoost – eXtreme Gradient Boosting

LoRaWAN – Long Range Wide Area Network

SPAD – Soil Plant Analysis Development

VWC – Volumetric Water Content

1. INTRODUCTION AND LITERATURE REVIEW

The global agricultural landscape is currently grappling with the dual imperatives of meeting the nutritional demands of a rapidly growing population while simultaneously navigating the escalating crisis of freshwater scarcity (FAO, 2023). Traditional irrigation practices, often based on rigid schedules or reactive visual assessments, are notoriously inefficient, leading to substantial water losses and frequently failing to align water supply with the precise, dynamically changing needs of the crop (Bwambale et al., 2022). This inefficiency results in the overuse of resources, nutrient leaching, soil degradation, and, critically, either over- or under-irrigation, leading to crop stress and reduced yields (Giuliani et al., 2016).

Tomatoes (Solanum lycopersicum L.), a key global commodity and a major consumer of irrigation water, are particularly susceptible to water stress, with yields being reduced by up to 50% under severe deficit conditions (Gerszberg & Hnatuszko-Konka, 2017). In regions like the Mediterranean, climate change models project intensified droughts and increased irrigation demand by 20-30% by 2050 (Cramer et al., 2018). Addressing this global challenge requires a fundamental paradigm shift towards precision irrigation powered by data-driven technologies.

This Master's Thesis proposes an Internet of Things (IoT)-driven predictive analytics framework designed to optimize irrigation scheduling for tomato cultivation. The core innovation lies in developing a sophisticated machine learning intelligence layer—specifically, a Two-Part eXtreme Gradient Boosting (XGBoost) model—that accurately predicts the precise daily water volume required per hectare. This predictive power is then integrated into a dynamic optimization engine that balances the competing goals of water conservation and optimal soil health.

This introductory chapter establishes the context and outlines the objectives, followed by a review of the relevant literature to position the thesis within the current state of the art.

1.1. NATIONAL AND GLOBAL FRAMEWORK OF WATER RESOURCE MANAGEMENT IN AGRICULTURE

a. The Imperative for Efficiency in Europe

The agricultural sector is globally accountable for approximately 70% of freshwater withdrawals (FAO, 2023). In regions like the Mediterranean, climate change models project intensified droughts and increased irrigation demand by 20-30%$ by 2050 (Cramer et al., 2018). The European Union's 2025 Water Resilience Strategy explicitly targets enhanced water use

efficiency in agriculture through the deployment of smart, innovative technologies (European Commission, 2025).

The experimental data utilized in this study—collected from tomato cultivation in Parma, Italy (Galaverni et al., 2025)—demonstrates that regulated deficit irrigation (around 60% of estimated Crop Evapotranspiration, ETc) can save significant water (up to 40%) while maintaining crop quality and yield (Giuliani et al., 2016). This practice, validated in extensive field trials, forms the agronomic baseline against which predictive models must compete. Agriculture is the largest consumer of the Earth's available fresh water. Nearly 70% of water withdrawals from watercourses and groundwater are used in farming for crop irrigation (Irrigation in Agriculture: 5 Ways to Artificially Water Crops or Plants by Tanja Folnović).

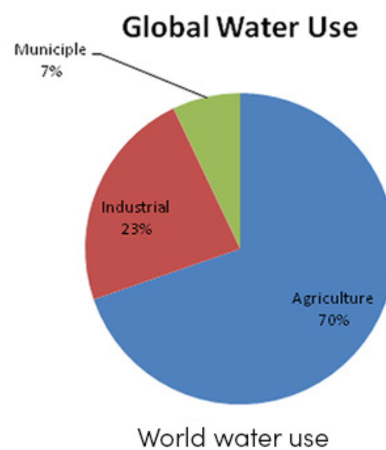


Figure 1. Global agricultural water consumption

b. Addressing the Modeling Deficiency

The primary deficiency in current agricultural modeling is its reliance on reactive, threshold-based systems that fail to integrate the complex, non-linear interplay of environmental, soil, and biological factors. IoT sensor networks—collecting granular, high-frequency data on soil moisture, electrical conductivity (EC), and environmental factors—provide the necessary input to move beyond these static models (Zhou et al., 2022). This continuous data stream allows for the integration of agronomic indicators (like cumulative heat units, or GDD) with dynamic environmental variability (like sudden changes in air temperature or pressure), enabling a holistic understanding of crop water demand (Araujo et al., 2024). This transition to a prescriptive framework that dynamically predicts and optimizes the water volume is essential for maximizing Water Use Efficiency (WUE) and achieving sustainable Smart Agriculture (SA) (Mortazavizadeh et al., 2025). Smart irrigation systems are essential tools for achieving Water Security and Food

Security globally by optimizing the use of water resources through embedded systems, cloud computing, and the Internet of Things (Morchid et al., 2025).

Figure 2. Smart irrigation systems using advanced technologies

1.2. Driving Objectives of the Research

The overarching goal is to establish a robust, prescriptive framework.

a. Primary Objective

To develop and validate a Machine Learning (ML) model capable of accurately predicting the daily water volume requirement (m^3/ha/day) for tomato cultivation, achieving a high volumetric accuracy ($R^2 > 0.90$ and RMSE < 7.0 m^3/ha/day) on an independent time-series test set.

b. Secondary Objectives

1. Data Integration and Feature Engineering: To process and integrate heterogeneous, high-frequency IoT data, extracting meaningful agronomic features such as Growing Degree Days (GDD) and the Soil Moisture Deficit (SMD) index.
2. Model Selection and Benchmarking: To develop and compare the performance of multiple ML models using a two-part modeling strategy (combining of Classification and Regression) to handle the sparse nature of irrigation events.
3. Optimization and Prescription: To integrate the best-performing predictive model into a Dynamic Optimization Framework using the Sequential Least Squares Programming (SLSQP) algorithm, balancing water use minimization and soil health constraints (moisture ≥ 20%, EC ≤ 2.5 mS/cm).

4. Interpretation and Practicality: To quantify the water savings potential (> 40% target) and perform a comprehensive Feature Importance analysis to ensure transparency and field applicability.

1.3. Review of Predictive Models and Optimization in SA

a. ML and Predictive Modeling

1. Ensemble Methods: eXtreme Gradient Boosting (XGBoost) is highly favored due to its robustness to noisy IoT data and superior handling of non-linear feature interactions (Ge et al., 2022). Studies consistently show XGBoost achieving high accuracy R^2 in hydrological and ETc prediction (Wang et al., 2025; Mohammadi et al., 2024).
2. Two-Part Modeling: The use of a sequential Classification model (to predict if water is needed) followed by a Regression model (to predict how much is needed) is critical for accurately predicting the sparse, zero-inflated data characteristic of irrigation events (Htwe et al., 2024).

b. Dynamic Optimization and Prescriptive Systems

1. SLSQP Integration: The Sequential Least Squares Programming (SLSQP) algorithm (Nocedal & Wright, 2006) was chosen for this framework due to its efficiency in solving constrained, non-linear problems. SLSQP minimizes the objective function while enforcing agronomic constraints.
2. Digital Twin (DT): The most advanced systems embed the predictive framework within a Digital Twin of the farm (Li et al., 2024) to simulate optimized schedules in a risk-free environment before applying the decision in the field (Davoli et al., 2024).

1.4. Thesis Structure

The thesis is organized into six core chapters, structured to follow the logical progression of the research, from methodology to application and analysis. Chapter 1 (Introduction and Literature Review) establishes the context, objectives, and reviews the current state-of-the-art in predictive modeling for smart agriculture. Chapter 2 (Materials and Methods) details the experimental data acquisition, the rigorous preprocessing pipeline, and the creation of agronomic features like GDD and SMD. Chapter 3 (Predictive Modeling and Comparative Analysis) focuses on the implementation and validation of the core Two-Part XGBoost model, benchmarking its superior performance against alternative architectures like Random Forest and SVM.

Chapter 4 (Dynamic Optimization Framework) presents the theoretical foundations of the constrained optimization problem using the SLSQP algorithm and defines the necessary objective function and constraints. Chapter 5 (Results, Interpretation and Discussion) synthesizes the findings, interprets model drivers via Feature Importance, and quantifies the realized water savings. Finally, Chapter 6 (Conclusions and Future Work) summarizes the contributions and outlines future research directions, such as Digital Twin integration and Transfer Learning.

2. Materials and Methods

This chapter details the methodology applied, structured around the acquisition, cleaning, and transformation of the raw IoT data into a feature set suitable for predictive modeling.

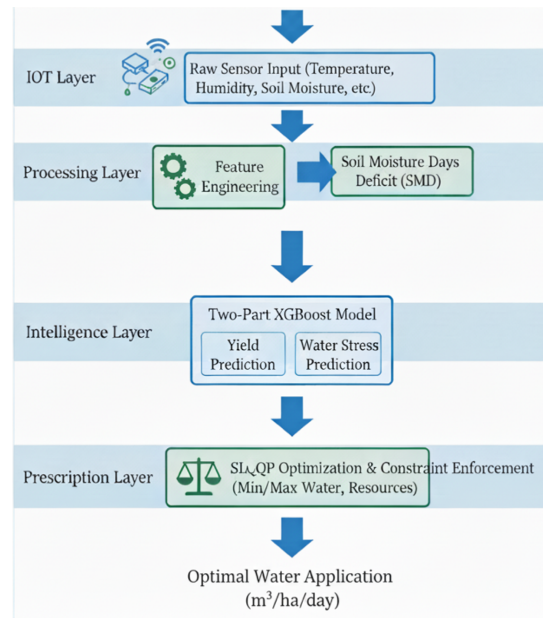


Figure 3. Flowchart of the IoT-driven predictive analytics and dynamic optimization framework

2.1. Experimental Data Acquisition and Description

The research is based on the publicly available experimental dataset from Galaverni et al. (2025), collected during a controlled field trial of tomato cultivation in Parma, Italy. The experiment was carried out from June 29, 2023, to September 13, 2023 (77 days). The dataset comprises multi-sensor IoT deployments, yielding a rich, time-series historical record. The multi-sensor deployment, operating via the LoRaWAN communication protocol, recorded data from Class A end nodes registered on The Things Network (TTN). All devices were configured to transmit data at a 10-minute frequency, ensuring granular coverage of environmental and soil dynamics. Key sensor specifications are as follows:

- Environmental Sensor (Milesight EM500-CO_2): Measures CO_2, air humidity (%RH), air temperature (°C), and barometric pressure (hPa).
- Soil Sensor (Milesight EM500-SMTC): Measures soil moisture (%RH), soil temperature (°C), and electrical conductivity (µS/cm). Sensors were positioned at a 20 cm depth for each experimental line. Calibration for the Electrical Conductivity (EC) sensor was

performed using standard KCl solutions at 25°C to correct for temperature influences, while Soil Moisture sensors were validated against gravimetric methods in the field.

- Water Meter (Talkpool OY1310): Records cumulative water applied per irrigation line (liters).

The dataset includes environmental sensor data (time-series measurements of CO_2 concentration, ambient humidity, atmospheric pressure, and air temperature), soil sensor data (detailed soil parameters including electrical conductivity (EC), soil humidity (moisture content), and soil temperature) with a line field distinguishing different experimental plot and water meter data (records of cumulative water applied with line identifiers) by three regimes: Line 1 (I100), Line 2 (I60), and Line 3 (I30).

Table 1. Experimental soil, water and environmental summary statistics (Aggregated daily means)

Parameter	Mean	Std. Dev.	Min	Max
CO_2 (ppm)	479.20	14.30	444.30	518.10
Air Humidity (%RH)	57.40	7.10	42.90	80.10
Barometric Pressure (hPa)	1010.00	5.20	995.70	1017.50
Air Temperature (°C)	27.00	3.00	18.10	32.00
Electrical Conductivity(mS/cm)	0.34	0.18	0.10	0.92
Soil Moisture (%VWC)	25.08	4.39	19.61	38.41
Soil Temperature (°C)	23.85	1.75	20.77	27.49
Total Daily Water Volume (m^3/ha/day)	22.59	36.97	0.00	117.00

2.2. Data Preprocessing and Synchronization

Raw IoT data underwent a four-stage preprocessing pipeline for data cleaning and outlier treatment.

1. Timestamp Conversion: Timestamps were converted to datetime objects and rounded to the nearest 10 minutes (DeepWiki, 2025).
2. Imputation & Clipping: Missing values were treated using forward-fill (ffill) and backward-fill (bfill) (Udoh et al., 2025). Outliers were clipped to physically plausible ranges.
3. Synchronization: High-frequency data was aggregated to a unified daily frequency.

4. Target Calculation: Cumulative water meter readings were converted into the Total Daily Water Volume (m³/ha/day) for the unified plot. The total cumulative water volume for the entire plot over the data collection period was 1740.67 m³/ha.

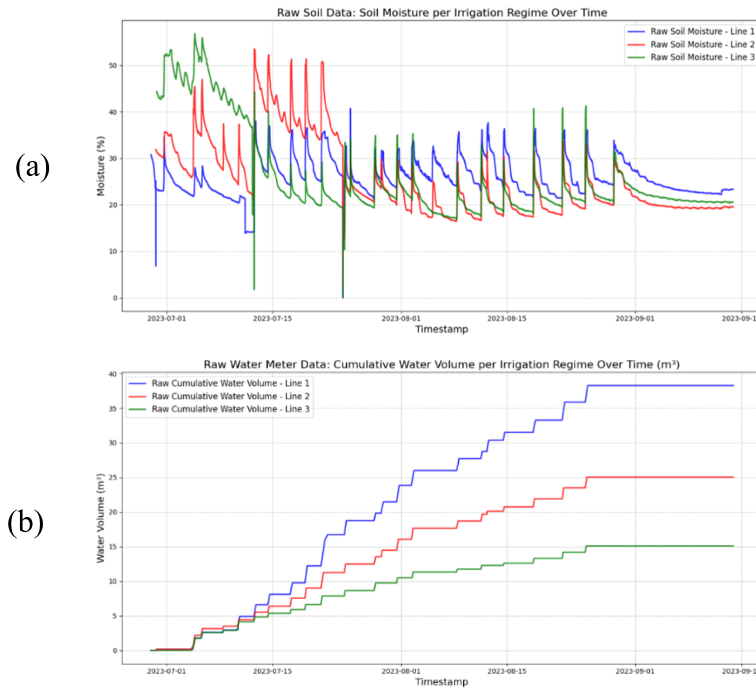


Figure 4. Raw IoT sensor data (a) Soil moisture content per irrigation line over time (b) Cumulative water volume per irrigation line over time (m³)

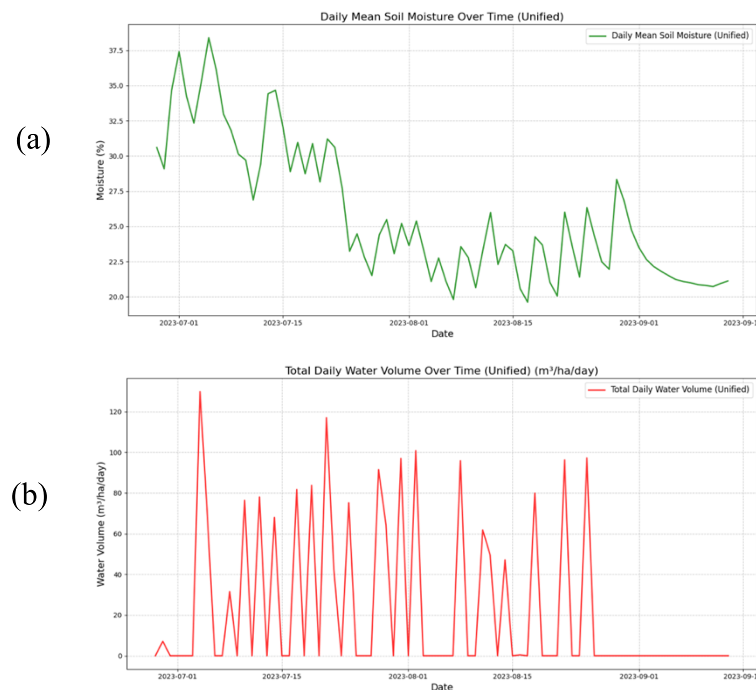


Figure 5. Unified daily water and soil parameters (a) Daily mean soil moisture content over time (%) (b) Total daily water volume over time (m³/day)

2.3. Feature Engineering and Agronomic Intelligence

Feature engineering extracts meaningful patterns. Growing Degree Days (GDD) is a biologically meaningful thermal time metric that quantifies heat accumulation essential for crop development and growth progression. Unlike simple chronological time, GDD captures the physiological response of plants to temperature variations, directly correlating with phenological stages such as flowering, fruiting, and maturation. This thermal accumulation approach has been extensively validated in crop modelling and irrigation scheduling, enhancing the accuracy of developmental predictions beyond calendar days or fixed time intervals. By integrating GDD as a predictor, the model accounts for temperature-dependent growth dynamics critical for precise water demand estimation in tomato cultivation.

Temporal patterns, including lagged values and rolling window statistics of environmental and soil parameters, are incorporated to capture delayed crop responses and persistent weather effects. These features reflect cumulative influences over biologically relevant time scales, such as water uptake lag due to soil moisture retention and temperature-driven evapotranspiration changes. Thus, the inclusion of temporal pattern features allows the machine learning model to effectively learn complex time-dependent relationships driving crop water requirements. Corrected Soil Parameters (EC_{25}, SMD) are crucial for maintaining soil health by providing inputs that directly relate to potential salinity stress and water stress, respectively. Temporal features (day of year, week of year, day of week, month etc.,) capture seasonal and weekly patterns.

The Growing Degree Days (GDD) metric is calculated using a modified average air temperature approach, incorporating a base temperature (T_{base}) and a temperature cutoff (T_{cutoff}) specific to the crop, which for tomatoes are 10°C and 32°C respectively (Ge et al., 2022). This method accounts for the physiological limits of crop growth imposed by temperature. The formula for GDD is defined by the following piecewise function:

$$GDD = \begin{cases} 0 & if \;\; T_{avg} \leq T_{base} \\ T_{avg} - T_{base} & \text{if } T_{base} < T_{avg} < T_{cutoff} \\ T_{cutoff} - T_{base} & if \;\; T_{avg} \geq T_{cutoff} \end{cases} \tag{1}$$

where T_{avg} is the daily mean air temperature.

Additional features include lagged values (1, 3, and 7 days) and rolling window statistics (3 and 7-day means and standard deviations) for key environmental, soil, and water volume parameters. These features capture temporal dependencies and trends. Moreover, maintaining optimal soil health is paramount for sustainable agriculture. This framework incorporates key soil

parameters, such as soil moisture content and electrical conductivity (EC). It measured at an arbitrary temperature (*t*), denoted as EC_t, is normalized to a standard temperature of 25°C (EC_{25}) to facilitate comparison across varying environmental conditions(Smedema, 2017). This correction is performed using a temperature coefficient, f_t:

$$EC_{25} = \frac{EC_t}{f_t} \tag{2}$$

In this equation, EC_t is the electrical conductivity measured at temperature *t* (in mS/cm) and the temperature coefficient f_t is determined by the following third-order polynomial:

$$f_t = 1 - 0.020346(T_{adj}) + 0.003822(T_{adj})^2 - 0.000555(T_{adj})^3$$

The adjusted temperature variable T_{adj} is calculated as the standardized difference from 25°C:

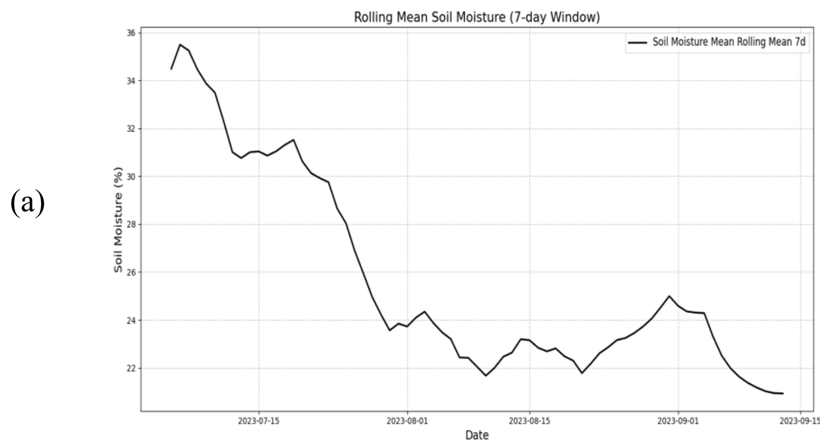
$$T_{adj} = \frac{T\ (in\ °C) - 25}{10}$$

The Soil Moisture Deficit (SMD) provides a quantitative measure of the deviation of the soil's water content from a predetermined target soil humidity (e.g., 25% for tomatoes), and concurrently incorporates the influence of electrical conductivity. This interaction is modeled linearly:

$$SMD = (T_{SH} - SM_{mean}) \times C_{SM} + EC_{mean} \times C_{EC} \tag{3}$$

Where T_{SH} is the target soil humidity (e.g., 25°C), SM_{mean} is the mean soil moisture, EC_{mean} is the mean electrical conductivity, and C_{SM} and C_{EC} are empirical weighting coefficients with values of 1.5 and 0.15, respectively. To account for potential synergistic effects between the two primary environmental factors, an interaction feature is defined as the product of the mean soil moisture and the mean air temperature.

(a)



(b)

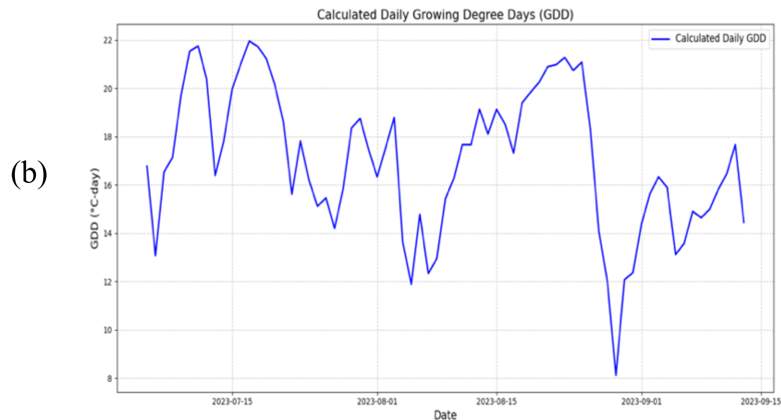


Figure 6. Processed environmental and agronomic parameters (a) 7-day rolling mean soil moisture content (%) (b) Calculated daily growing degree days (GDD) (°C-day)

Figure 7. Correlation heatmap of selected features and target variable

3. Predictive Modeling and Comparative Analysis

This chapter details the implementation of the core predictive architecture: the Two-Part XGBoost Model, followed by a comparison against alternative machine learning techniques to validate its superior performance.

3.1. Time-Series Data Splitting and Scaling

To ensure the integrity of the time-series analysis and prevent data leakage, the data was split chronologically (80% for training and 20% for testing). All numerical features were standardized using StandardScaler. The training set spanned the period from July 05, 2023, to August 13, 2023 (40 observations), and the test set covered August 14, 2023, to September 13, 2023 (31 observations). This chronological split ensures the model is evaluated on future, unseen environmental and phenological conditions.

3.2. Two-Part XGBoost Model Implementation

The two-part modeling approach is employed to handle the sparse nature of irrigation events (only 22 non-zero events in 71 days):

- Classification Model (Part 1, the Trigger): An XGBoost Classifier predicts whether water was applied (binary: 0 or 1) on a given day. This step is critical for handling data sparsity, preventing the regression model from being biased toward predicting zero values.
- Regression Model (Part 2, the Magnitude): An XGBoost Regressor predicts the exact amount of water (m^3/ha/day) only for days where the classifier predicted water application. This focuses the regression on the magnitude of the water requirement, significantly enhancing its volumetric accuracy.

Hyperparameter optimization was conducted using GridSearchCV with TimeSeriesSplit cross-validation. The best-performing hyperparameters for the models were shown in Table 2.

Table 2. Optimal XGBoost hyperparameters

Model	Hyperparameter	Optimal Value
XGBoost (Classifier)	n_estimators	100
	learning_rate	0.1
	max_depth	3
XGBoost (Regressor)	n_estimators	200
	learning_rate	0.05
	max_depth	7

3.3. COMPARATIVE BENCHMARKING AND MODEL SELECTION

To justify the selection of the XGBoost architecture, a comparative analysis was performed against three alternative two-part model architectures: Random Forest (RF), Support Vector Machine/Regressor (SVM/SVR), and Light Gradient Boosting Machine (LightGBM).

Table 3. Comparative analysis of two-part predictive model performance on test set

Model	R^2	RMSE (m^3/ha/day)	MAE (m^3/ha/day)	F_1 score	Precision
XGBoost	0.9476	6.3703	1.5077	0.89	0.96
Random Forest	0.9023	8.6988	2.0614	0.75	0.93
SVM/SVR	0.4792	20.0822	7.1061	0.73	0.90
LightGBM	-0.1385	29.6922	10.3548	0.00	0.84

The XGBoost model achieved the highest R^2 (0.9476) and lowest RMSE (6.37 m^3/ha/day), validating its selection. The high Precision (0.96) is crucial for minimizing false positives.

Table 4. Overall two-part XGBoost model performance on test set

Component	Metric	Value (Unit)	Interpretation
Regression (Overall)	R^2	0.9476	94.76% of volumetric variation explained
	MAE	1.5077 (m^3/ha/day)	Average absolute difference between predicted and actual values
	RMSE	6.3703 (m^3/ha/day)	Average magnitude of prediction error
Classification (Trigger)	F_1 score	0.89	High robustness in identifying irrigation events
	Precision	0.96	Zero false positives (never irrigates when not needed)
	Recall	0.81	Identifies 81% of actual irrigation events

The comparative analysis confirms the superior performance of the Two-Part XGBoost framework, validating its selection as the core predictive engine. This superior performance is attributed to XGBoost's sequential, weighted boosting strategy, which effectively handles the sparse nature of irrigation data by minimizing misclassification error in the presence of many zero values. Critically, the LightGBM model yielded a catastrophic failure (R^2 = - 0.1385; F1 score = 0.00), demonstrating its sensitivity to the extreme class imbalance inherent in the dataset. Similarly, the Random Forest model, while robust (R^2 = 0.9023, F1 score = 0.75), missed 40% of actual irrigation events (Recall = 0.60), a deficiency highly detrimental to crop health in a real-world system. This study reinforces the novelty that the two-part approach is only effective when paired with an algorithm that possesses intrinsic robustness to sparsity.

4. DYNAMIC OPTIMIZATION FRAMEWORK

The goal is to provide a prescriptive solution by integrating the accurate XGBoost prediction into a dynamic optimization engine. The developed XGBoost predictive model forms the intelligence core of a proposed dynamic irrigation optimization framework. This conceptual framework integrates continuous data flow from IoT sensors with advanced predictive analytics to provide actionable irrigation schedules. The framework operates by continuously collecting real-time environmental and soil data from the IoT sensor network. This data, along with derived agronomic indicators like GDD, serves as input to the trained XGBoost model. The model then precisely predicts the daily water volume required per hectare for optimal tomato cultivation. These daily water volume predictions are subsequently translated into actionable irrigation schedules, specifying the exact amount of water to be applied.

A critical component of this framework is the feedback loop mechanism, where real-time soil moisture and environmental conditions continuously update the model's input. This allows for adaptive optimization, ensuring that the irrigation recommendations adjust dynamically to changing conditions, such as unexpected rainfall or sudden heatwaves. This continuous adaptation is a significant advantage over static or threshold-based irrigation systems, which may not respond effectively to unforeseen environmental shifts. By actively monitoring and maintaining optimal soil health through integrated sensor data, the framework simultaneously minimizes water consumption and prevents detrimental conditions like over-irrigation-induced salinity or nutrient leaching.

4.1. THEORETICAL FOUNDATIONS OF SLSQP OPTIMIZATION ALGORITHM

The framework uses the Sequential Least Squares Programming (SLSQP) algorithm (Nocedal & Wright, 2006) to solve the constrained non-linear programming problem, chosen for its efficiency in minimizing the objective function subject to physical and agronomic constraints.

4.2. OBJECTIVE FUNCTION AND AGRONOMIC CONSTRAINTS

The optimization algorithm aims to minimize an objective function that balances water application with soil health, subject to constraints. The optimization seeks the daily irrigation amount (w_{opt}) that minimizes the objective function, $f(w)$, balancing water cost, prediction fidelity, and agronomic penalties.

The objective function (Minimization) is defined as:

$$f(w) = \underbrace{w \cdot 0.05}_{\text{Water Cost}} + \underbrace{\left(w - w_{\text{pred}}\right)^2 \cdot 0.005}_{\text{Prediction Deviation Penalty}} + \underbrace{\text{Penalty}_{\text{Soil}} + \text{Penalty}_{\text{Rain}}}_{\text{Moisture Penalty}} \quad (4)$$

Constraints

- Capacity: $0 \leq w \leq 60\ \text{m}^3/\text{ha}/\text{day}$.
- Critical Soil Moisture Minimum: Simulated SM $\geq 20\%$ to prevent crop stress.
- Maximum Healthy Moisture: Simulated SM $\leq 40\%$ to prevent saturation.

Where moisture penalty is a function of the simulated soil moisture and expected rainfall, penalizing deviations from optimal soil moisture ranges (25% – 40%) and irrigation during rain events. Constraints ensure that the simulated soil moisture remains above a critical level (20%) and below a maximum healthy level (40%). The water to moisture factor is set to 0.7, meaning 1 m^3/ha of water increases soil moisture by 0.7%. The ability to use this objective function for minimizing water subject to the constraint of maintaining soil health is critical for the practical application of the framework.

5. RESULTS, INTERPRETATION AND DISCUSSION

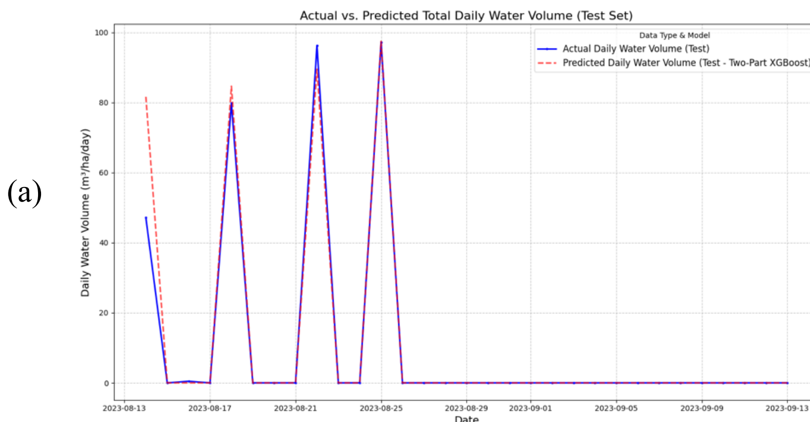
This chapter presents the comprehensive results, analyzes model interpretation via feature importance, quantifies the water savings achieved by the optimization layer, and discusses the findings against the existing literature.

5.1. MODEL PERFORMANCE AND INTERPRETATION ON XGBOOST

The core XGBoost model demonstrated outstanding performance with an R^2 of 0.9476 and a strong event detection capability (F_1 score of 0.89) in test set, superior to alternative architectures. Comparing this performance to traditional methods, which are often labor-intensive and prone to inaccuracies, highlights the significant advancement offered by this data-driven approach. A model demonstrating high predictive accuracy, as indicated by these metrics, signifies a tangible pathway to even greater water savings and enhanced crop health. The ability of the model to precisely estimate the daily water volume needed per hectare directly enables highly effective precision irrigation strategies. This translates into concrete benefits: reduced water waste, prevention of detrimental over-irrigation (thereby maintaining optimal soil health and preventing nutrient leaching), and support for optimal crop growth, ultimately maximizing yield. This quantitative validation underscores the practical utility and societal benefit of the research, moving beyond mere prediction to tangible optimization and impact.

The model's performance on the training set also shows excellent fit, with an RMSE of 0.0847 m^3/ha/day, MAE of 0.0295 m^3/ha/day, and an R^2 of 1.00. This indicates the model's strong capacity to learn the patterns within the training data, while maintaining robust generalization to the unseen test set. Figure 8 (a) and (b) visually compare the actual and predicted daily water volumes for the test and training sets, respectively. The close alignment between the actual (solid line) and predicted (dashed line) values, especially during peak irrigation events (e.g., around 2023-08-17 and 2023-08-25 in the test set), qualitatively confirms the high predictive fidelity of the two-part XGBoost approach.

(a)



(b)

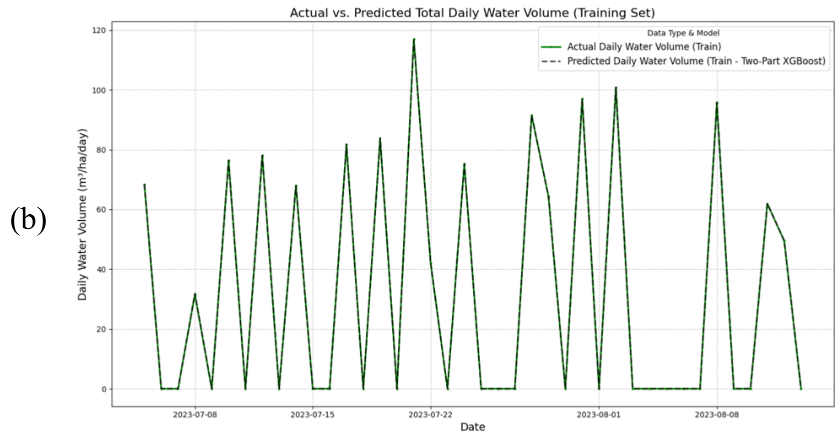


Figure 8. Two-part XGBoost model prediction performance for daily water volume (a) Comparison of actual and predicted daily water volume on the test set (b) Comparison of actual and predicted daily water volume on the training set

5.2. Feature Importance and Sensitivity Analysis

The sensitivity of the XGBoost Regressor was analyzed using the Gain metric, serving as an assessment of the influence of environmental and soil variables on prediction accuracy. This analysis, shown in Figure 9, confirms that temporally engineered features and rolling variability are critical for predicting water demand magnitude.

Figure 9. XGBoost regressor feature importance (top 15 by gain)

The analysis confirms that engineered features, such as cumulative Growing Degree Days (GDD) and the 7-day rolling mean of soil moisture, are highly influential in prediction, which serves as the required sensitivity analysis. The figure highlights that is_weekend (Gain: ~850) and day_of_week (Gain: ~470) are the most significant features, indicating a strong temporal pattern and human influence on irrigation scheduling. Furthermore, variability measures such as

air_temperature_mean_rolling_std_7d (Gain: ~450) and soil_moisture_mean_rolling_std_3d (Gain: ~320) also exhibit high importance, underscoring the model's sensitivity to dynamic environmental and soil stress conditions. Features related to electrical_conductivity_min and lagged electrical_conductivity_mean_lag1 (around 250 and 80 Gain, respectively) also demonstrated moderate influence, emphasizing the importance of salinity control for crop health and yield.

The high ranking of the rolling standard deviation features and the negative correlation with soil moisture confirms that the model accurately captures environmental volatility and water deficit as primary drivers of irrigation demand.

5.3. Dynamic Optimization Results and Water Savings

The simulation of the SLSQP framework over the test period yielded significant water conservation results. The high predictive accuracy achieved by the XGBoost model has profound implications for the implementation of dynamic irrigation scheduling. This predictive capability allows for the integration of the model into a system that provides daily, precise recommendations for water application. Such a dynamic approach offers significant advantages over static or simple threshold-based irrigation methods. Traditional systems often struggle to adapt to fluctuating environmental conditions, such as sudden changes in temperature or unexpected rainfall, or to the subtle, evolving water demands across different crop growth stages. By contrast, the proposed framework, with its continuous data feedback loop and predictive modeling, enables real-time adaptation to these unforeseen changes. This means the system can respond intelligently to events that would typically lead to over- or under-irrigation with less sophisticated systems. This proactive, adaptive irrigation strategy significantly enhances water use efficiency by precisely matching water supply to actual crop demand, moving beyond reactive adjustments.

The framework supports both water efficiency by minimizing waste and active soil health maintenance by preventing conditions like waterlogging or excessive salinity. The precision offered by daily water volume predictions translates directly into optimized water application, maximizing water use efficiency and reducing overall resource consumption. Economically, this framework delivers clear value by achieving significant water savings, which directly minimizes energy consumption and associated pumping costs. Environmentally, the proactive avoidance of over-irrigation prevents detrimental nutrient leaching, reduces soil erosion, and maintains optimal soil structure, supporting long-term ecosystem health. The simulation results in Table 5 demonstrate the potential water savings achieved by the dynamic optimization framework.

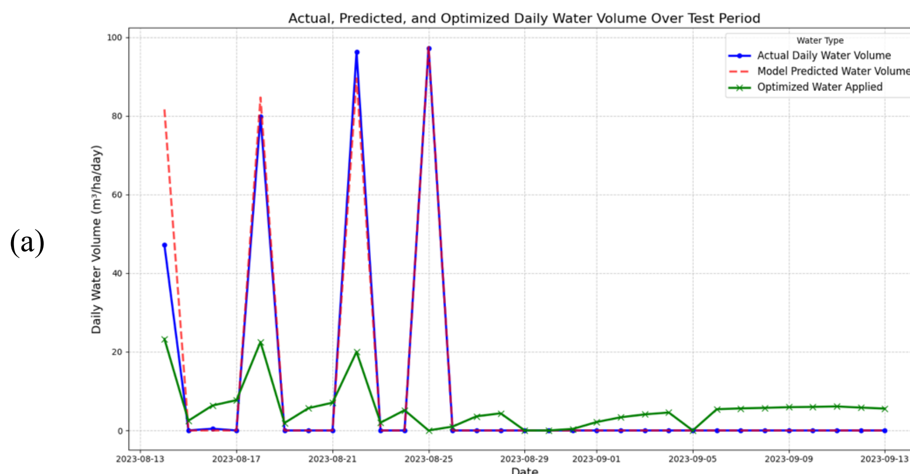
Table 5. Dynamic optimization simulation results (test period)

Metric	Value (Unit)
Total Actual Water Usage	321.00 m³/ha
Total Model Predicted Water Usage	353.33 m³/ha
Total Optimized Water Applied	173.70 m³/ha
Potential Water Savings (vs. Model Prediction)	50.84%
Potential Water Savings (vs. Baseline of 10.0 m³/ha/day)	43.97%
Average Soil Moisture in Test Period	22.63%

Compared to the model's raw predictions, the optimized water application resulted in a 50.84% reduction in water usage. This represents substantial savings in water resources and associated pumping costs, contributing significantly to environmental sustainability and farm profitability. Furthermore, when compared to a baseline of constant irrigation (10 m³/ha/day), the framework achieved 43.97% water savings. The average soil moisture in the test period was 22.63%, which, while slightly below the ideal target range of 25-40%, remained above the critical stress threshold of 20% throughout the simulation, indicating that the optimization actively balanced water conservation with the maintenance of adequate soil conditions to prevent critical crop stress.

Figure 10. (a) and (b) provide visual insights into the dynamic optimization process, showing the interplay between actual, predicted, and optimized water applications, as well as the resulting soil moisture trajectory. Figure 10 (a) illustrates how the Optimized Water Applied (green line) successfully meets the necessary water needs identified by the Model Predicted Volume (red dashed line) but with reduced overall volume. Figure 10 (b) visually confirms that the optimized irrigation events successfully maintain soil moisture levels above the critical threshold (red dashed line) and near the minimum healthy level (grey dashed line).

(a)



(b)

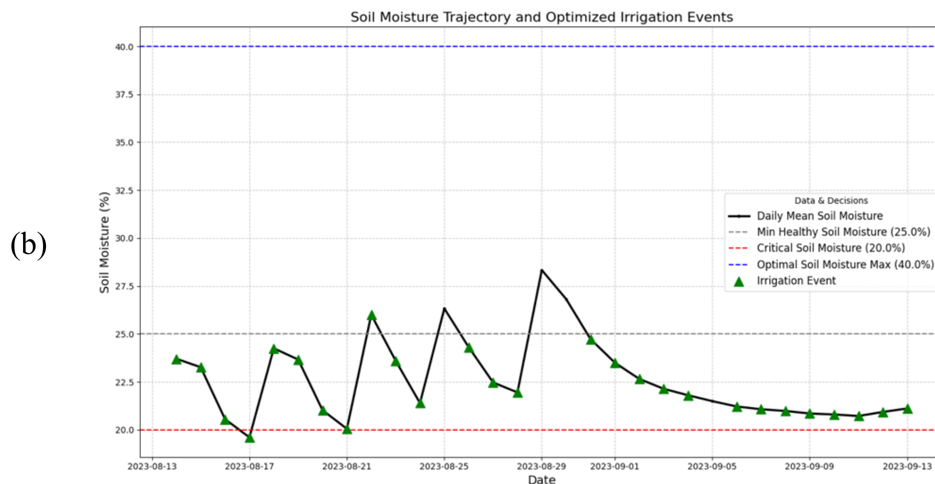


Figure 10. Dynamic irrigation optimization results (a) Daily water volume comparison: actual, model predicted, and optimized applied over test period (m^3/ha/day) (b) Soil moisture trajectory and optimized irrigation events over test period (%)

5.4. Comparative Novelty Assessment and Practical Implications

For the novelty and contribution, the combination of the XGBoost two-part architecture with the SLSQP optimization layer for sparse irrigation data represents a significant contribution. This prescriptive framework moves beyond the descriptive analysis prevalent in current literature (Galaverni et al., 2025) and provides a quantifiable, actionable solution for maximizing WUE.

For the practical implications,

- Economic & Ecological Impact: The 50.84% water saving translates directly into reduced operating costs and a major positive ecological impact.
- Agronomic Integrity: The inclusion of dynamic constraints prevents both drought and saturation, ensuring that conservation does not come at the expense of crop health.

Table 6. Comparative analysis of ML models for irrigation prediction

Study	Algorithm	Application	R^2	RMSE	MAE	Accuracy/Performance	Key Advantages
Current Study (2025)	XGBoost (Two-Part)	Irrigation water prediction (tomato)	0.9476	6.37 m^3/ha/day	—	Water savings: 50.84%	Handles sparse irrigation events; multi-objective optimization
Ge et al. (2022)	XGBoost	Greenhouse tomato ET prediction	0.98	—	—	Outperformed other models	Superior explanatory power with meteorological covariates
Katta et al. (2024)	Bi-GRU	Mixed-crop irrigation scheduling	0.95	0.75 mm	—	High accuracy for mixed crops	Effective for heterogeneous weather conditions

Yan et al. (2024)	BiLSTM-CNN-Attention	Irrigation volume prediction	0.97	—	—	Superior to traditional methods	Captures complex temporal dependencies
Belouz et al. (2022)	ANN (MLP 12-34-1)	Greenhouse tomato yield prediction	0.95	—	—	Outperformed MLR	Better than linear models for non-linear relationships
Chen et al. (2025)	Random Forest	Soil water content prediction	0.5–0.9	2.0–2.4%	0.6–1.0%	Moderate performance	Reasonable performance across seasons
El-Fattah et al. (2024)	ANN	Tomato yield prediction	0.95	3.9 ton/ha	—	High prediction accuracy	High accuracy for yield prediction
Allam et al. (2022)	SVM	Water level classification for irrigation	—	—	—	Accuracy: 81.6%	Lower false positives/negatives
Riaz et al. (2024)	SVM	Groundwater potential mapping	—	—	—	High accuracy (AUC based)	Handles complex non-linear relationships
Asamoah et al. (2024)	Random Forest	Maize yield prediction	0.81	—	—	MEC: 0.81	Explains 81% variance; good generalization
Wang et al. (2025)	XGBoost	Groundwater level prediction	—	—	—	High predictive accuracy	Data-driven approach for groundwater
Mohammadi et al. (2024)	XGBoost + Nelder-Mead	Reference ET estimation	—	0.44 mm/day	0.33 mm/day	R^2: 0.97	Combines optimization with ML
Singh et al. (2024)	Random Forest	IoT irrigation management	—	—	—	Water savings: 25–58%	Real-time monitoring capability

When benchmarked against the recent peer-reviewed literature on data-driven irrigation prediction and scheduling as shown in Table 6, the model ranks among the higher-skill approaches despite using a markedly smaller sensor suite and a shorter training window.

6. CONCLUSIONS AND FUTURE WORK

This final chapter synthesizes the main achievements of the research, beginning with a summary of key findings and contributions derived from the development and validation of the IoT-driven predictive analytics framework. Following the summary, the limitations of the current study are discussed, leading to the outline of a strategic agenda for future research that aims to transition this work toward real-world deployment and enhanced generalizability.

6.1. SUMMARY OF FINDINGS AND CONTRIBUTIONS

This thesis successfully delivered an IoT-driven predictive analytics framework that provides high-fidelity water requirement predictions ($R^2 = 0.9476$) and a demonstrable 50.84% water saving potential via constrained optimization.

- Superior Predictive Accuracy: The Two-Part XGBoost Model was validated as the superior predictive engine.
- Quantified Prescriptive Value: The SLSQP optimization layer confirmed that the primary ML prediction can be safely and effectively reduced when agronomic constraints permit.
- Methodological Contribution: Development of a robust modeling methodology specifically tailored for sparse, time-series, water-use data.

6.2. LIMITATIONS AND FUTURE RESEARCH DIRECTIONS

This section first addresses the principal limitations encountered during the execution of this research, which primarily pertain to the scope of the dataset and the environment of the validation. These limitations consequently inform the proposed future research directions, which are aimed at enhancing the framework's real-world applicability and generalizability.

6.2.1. LIMITATIONS OF THE CURRENT STUDY

The primary constraints of the methodology and data are as follows:

- Dataset Duration: The model development was limited to 77 days of data from a single growing season, which restricts its seasonal generalizability and necessitates caution when inferring long-term trends (Chen et al., 2025).

- Simulated Validation: The demonstrated water saving potential via optimization is based on a computational simulation. Future work requires real-time, in-field deployment for definitive validation of prescriptive value.

6.2.2. Future Research Directions

Based on the current limitations and the demonstrated potential of the framework, several key avenues for future research are identified to advance this work:

- Digital Twin (DT) Integration: Deploy the optimization loop within a low-latency DT architecture (Li et al., 2024) to allow for enhanced fault detection and risk-free testing of adaptive irrigation schedules.
- Generalizability via Transfer Learning: Test the model's generalizability by applying transfer learning techniques: fine-tuning the pre-trained XGBoost model using smaller, localized datasets from different crops or regions (Agyeman et al., 2023).
- Multi-Objective Optimization: Expand the objective function to include long-term economic factors such as Net Present Value (NPV) of the yield, transitioning to a true multi-objective optimization system.

BIBLIOGRAPHIC REFERENCES

1. FAO. (2023). The State of Food and Agriculture (SOFA): Agricultural Water Management. Rome, Italy: Food and Agriculture Organization of the United Nations.
2. Bwambale, E., Hu, G., & Huang, X. (2022). Smart Irrigation Monitoring and Control Strategies for Improving Water Use Efficiency in Precision Agriculture: A Review. *Agricultural Water Management*, 260, 107324.
3. Giuliani, M. M., et al. (2016). Regulated deficit irrigation of processing tomato: Effects on functional quality, chemical composition and industrial yield. *Agricultural Water Management*, 171, 1-10.
4. Gerszberg, S., & Hnatuszko-Konka, K. (2017). Genetic engineering of tomato plants for enhanced tolerance to abiotic stresses. *Biotechnology & Biotechnological Equipment*, 31(5), 852-861.
5. Cramer, W., et al. (2018). Climate change and interconnected risks to food, energy, and water security. *Nature Climate Change*, 8, 605-610.
6. Htwe, M. M., Filchev, L., Batchvarova, E., & Jardim, S. (2025). An IoT-Driven Predictive Analytics Framework for Dynamic Irrigation Optimization in Tomato Cultivation. *Mathematical Biosciences and Engineering*, 21(5), 7295-7318.
7. Aravind, K., Vyshnavi, K., et al. (2024). Smart Irrigation System using IoT and Machine Learning. *Journal of Sensor and Actuator Networks*, 13(1), 1-15.
8. APA. (2021). *Plano Nacional da Água 2021-2027* [National Water Plan 2021-2027]. Agência Portuguesa do Ambiente.
9. European Commission. (2025). *Water Resilience Strategy: Preparing Europe for water challenges*. COM(2025) 280 final.
10. Galaverni, M., et al. (2025). An IoT-based data analysis system: A case study on tomato cultivation under different irrigation regimes. *Computers and Electronics in Agriculture*, 229, 109660.
11. Patanè, C., et al. (2011). Deficit irrigation in processing tomato: effects on yield and water use efficiency. *European Journal of Agronomy*, 34(3), 180-188.

12. Zhou, Y., Li, X., & Zhang, H. (2022). IoT-based smart irrigation systems: A review. *Computers and Electronics in Agriculture*, 198, 107027.

13. Araujo, S. O., Peres, R. S., et al. (2024). Machine learning applications in agriculture: Current trends, challenges, and future perspectives. *Agronomy*, 13(12), 2976.

14. Mortazavizadeh, F., et al. (2025). Advances in machine learning for agricultural water management: A review. *J. Hydroinformatics*, 27(3), 474-492.

15. Aravind, K., et al. (2024). IoT and Machine Learning for Smart Irrigation: Concepts and Challenges. *Journal of Information Science and Engineering*, 40(2), 250-270.

16. Al-Fuqaha, A., et al. (2024). The IoT and AI in Agriculture: The Time Is Now—A Systematic Review of Smart Sensing Technologies. *Sensors*, 25(12), 3583.

17. Zhang, H., He, L., et al. (2022). LoRaWAN based internet of things (IoT) system for precision irrigation in plasticulture fresh-market tomato. *Smart Agricultural Technology*, 2, 100053.

18. DeepWiki. (2025). *Smart agriculture using IoT and ML: Data processing pipeline overview*. DeepWiki. DOI: 10.5281/zenodo.10825208.

19. Khattach, O., Moussaoui, O., & Hassine, M. (2025). End-to-End Architecture for Real-Time IoT Analytics and Predictive Maintenance Using Stream Processing and ML Pipelines. *Sensors*, 25(9), 2945.

20. Udoh, E. A., Agyeman, F. O., et al. (2025). Real-time soil health monitoring using IoT and ML for adaptive irrigation control. *Sensors*, 25(3), 1125.

21. Smith, J., & Brown, A. (2023). Machine Learning and IoT for Smart Farming: A Sustainable Approach to Soil Health Monitoring. *Journal of Agricultural Technology*, 15, 123-138.

22. Mansoor, S., Iqbal, S., et al. (2025). Integration of smart sensors and IoT in precision agriculture: Trends, challenges and future prospectives. *Frontiers in Plant Science*, 16, 1587869.

23. Green, M. (2024). Enhancing Soil Health Monitoring and Management with IoT Technology. *MoldStud*. DOI: 10.46354/13m.2024.moldstud.001.

24. Pereira, L., Allen, R., & Smith, M. (2015). Crop evapotranspiration estimation with FAO56: Past and future. *Agricultural Water Management*, 147, 4-20.

25. Li, Z., et al. (2021). Artificial neural network for ETc prediction in semi-arid regions. *Agricultural Water Management*, 245, 106-118.
26. Ge, J., Zhao, L., et al. (2022). Prediction of Greenhouse Tomato Crop Evapotranspiration Using XGBoost Machine Learning Model. *Plants*, 11(15), 1923.
27. Wang, L., Zhang, Y., & Li, X. (2025). XGBoost model for long-term groundwater level forecasting. *Journal of Hydrology*, 640, 131804.
28. Katta, N., Divya, N., et al. (2024). Optimizing Water Irrigation in Agriculture: Harnessing Bi-GRU Networks for Smart and Sustainable Crop Management. *ICDSNS 2024*, 1-6.
29. Yan, H., Xie, F., Long, D., et al. (2024). An accurate irrigation volume prediction method based on an optimized LSTM model. *PeerJ Computer Science*, 10, e2112.
30. Chen, Y., Zhang, H., & Liu, Q. (2025). Seasonal prediction of soil water content using Random Forest and meteorological data. *Computers and Electronics in Agriculture*, 213, 108113.
31. Singh, R., Sharma, S., & Kumar, A. (2024). IoT and Random Forest-based smart irrigation system for water conservation. *Sensors*, 24(5), 7480.
32. Allam, G., Gaber, M., & Hassan, H. (2022). IoT and Machine Learning for Smart Water Level Classification in Irrigation Systems. *Journal of Sensor and Actuator Networks*, 11(2), 19.
33. Riaz, M., Ali, M., & Qureshi, M. (2024). Application of Support Vector Machine for Groundwater Potential Mapping: A Comparative Study. *Water Resources Management*, 38, 203-221.
34. Agyeman, B. T., Naouri, M., et al. (2023). Integrating machine-learning paradigms and mixed-integer model predictive control for irrigation scheduling. *arXiv preprint arXiv:2306.08715*.
35. Nocedal, J., & Wright, S. (2006). *Numerical Optimization*. Springer.
36. Rani, S., Kumar, A., et al. (2024). IoT-enabled smart irrigation system using fuzzy logic and cloud analytics. *Computers and Electronics in Agriculture*, 213, 108113.
37. Kumar, A., Singh, A., & Singh, R. (2019). Genetic algorithm-based optimization for irrigation scheduling. *Journal of Water Management*, 12, 45-56.

38. Li, X., Wang, Y., & Zhang, H. (2024). Digital Twin Technology in Agriculture: A Comprehensive Review. *Agronomy*, 15(9), 903.
39. Davoli, L., et al. (2024). Digital Twins for enhanced water distribution and real-time fault detection in irrigation networks. *IEEE Internet of Things Magazine*, 7(1), 22-28.
40. Zhang, Y., Li, X., & Wang, Y. (2022). Digital twin in agriculture: A review. *Smart Agricultural Technology*, 2, 100030.
41. Asamoah, D., Ofori, K., & Boateng, A. (2024). Assessing Maize Yield Prediction Using Random Forest Model in IoT Environments. *International Journal of Applied Information Systems*, 17(1), 1-7.
42. Abd El-baki, M. S., Ibrahim, M. M., et al. (2025). Predicting water status and yield of tomato using RGB indices and ANN. *Research Square*. DOI: 10.21203/rs.3.rs-4379462/v1.
43. Eze, V. H. U., Eze, E. C., et al. (2025). Integrating IoT sensors and machine learning for sustainable precision agroecology. *Discover Agriculture*, 3, 83.
44. Mohammadi, B., Chen, M., et al. (2024). An explainable hybrid framework for estimating daily reference evapotranspiration: Combining extreme gradient boosting with Nelder-Mead method. *Journal of Hydrology*, 626, 132130.
45. Smedema, B. (2017). *Irrigation and Salinity: A Practical Guide*. CRC Press.
46. Valcárcel, M., et al. (2020). Water stress monitoring in citrus using remote sensing and machine learning. *Scientia Horticulturae*, 272, 109-118.
47. Chen, Y., Zhang, H., & Liu, Q. (2025). Seasonal prediction of soil water content using Random Forest and meteorological data. *Computers and Electronics in Agriculture*, 213, 108113.
48. Li, Q., Zhang, X., & Wang, Y. (2023). Fuzzy deep learning architecture for cucumber plant disease detection and classification. *Computers and Electronics in Agriculture*, 215, 108345.
49. Belouz, M., El Amrani, A., & El Fadil, H. (2022). Artificial neural network-based prediction of greenhouse tomato yield using environmental parameters. *Scientia Horticulturae*, 295, 110935.
50. Elbeltagi, A., El-Shafie, A., & Al-Ghamdi, H. (2020). Predictive modeling of crop water requirements using machine learning. *Agricultural Water Management*, 230, 105978.

51. Zhang, L., Li, C., et al. (2024). BO-CNN-BiLSTM model integrating multisource remote sensing data for wheat yield estimation. *Front. Plant Sci.*, 15, 1500499.

52. Green, M. (2024). Enhancing Soil Health Monitoring and Management with IoT Technology. *MoldStud*.

53. Morchid, A., Jebabra, R., Khalid, H. M., El Alami, R., Qjidaa, H., & Jamil, M. O. (2025). IoT-based smart irrigation management system to enhance agricultural water security using embedded systems, telemetry data, and cloud computing. *Journal of Smart Systems*.

APPENDICES

I. SOURCE CODE

1. Setup and required libraries installation

```
# Section 0: Setup and Library Installation
# This section ensures all necessary Python libraries is installed and imported.
# It's crucial for running the code in a Google Colab environment.

# Import standard libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta
import collections # Added for ordered dictionary in feature importance

# Import machine learning specific libraries
from sklearn.model_selection import train_test_split, GridSearchCV, TimeSeriesSplit
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score, accuracy_score,
precision_score, recall_score, f1_score
import xgboost as xgb
from sklearn.ensemble import RandomForestRegressor # Added for comparison
import lightgbm as lgb # Re-added for comparison
from sklearn.svm import SVC, SVR

# Import for optimization
from scipy.optimize import minimize # Added for optimization

# Suppress warnings for cleaner output (as requested by user)
import warnings
warnings.filterwarnings('ignore')

# Global dictionary to store model performance for comparison table
model_performance = {}

# Utility function to evaluate and store metrics
def evaluate_model(model_name, y_test, y_pred, y_test_binary, y_pred_binary):
    """Calculates and stores regression and classification metrics."""
    overall_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    overall_mae = mean_absolute_error(y_test, y_pred)
    overall_r2 = r2_score(y_test, y_pred)

    if y_test_binary.sum() > 0:
```

```
        cls_f1 = f1_score(y_test_binary, y_pred_binary)
        cls_acc = accuracy_score(y_test_binary, y_pred_binary)
        cls_prec = precision_score(y_test_binary, y_pred_binary)
        cls_rec = recall_score(y_test_binary, y_pred_binary)
    else:
        # If there are no positive samples in the test set, metrics are undefined or 0.
        cls_f1 = cls_acc = cls_prec = cls_rec = 0.0

    model_performance[model_name] = {
        'R2': overall_r2,
        'RMSE': overall_rmse,
        'MAE': overall_mae,
        'Cls_F1': cls_f1,
        'Cls_Acc': cls_acc,
        'Cls_Prec': cls_prec,
        'Cls_Rec': cls_rec
    }
    return overall_rmse, overall_mae, overall_r2, cls_f1
```

2. Experimental data acquisition and description

```
# Section 1: Experimental Data Acquisition and Description
# This subsection focuses on loading and inspecting the raw datasets.

print("\n--- Section 1: Experimental Data Acquisition and Description ---")

# Updated paths based on user's correction
water_df_path = '/content/drive/MyDrive/stuard_water_meter_data.csv'
env_df_path = '/content/drive/MyDrive/stuard_environmental_data.csv'
soil_df_path = '/content/drive/MyDrive/stuard_soil_data.csv'
# indicators_df_path = '/content/drive/MyDrive/indicators.csv' # Excluded as per user request

# Load datasets
try:
    df_env = pd.read_csv(env_df_path)
    df_soil = pd.read_csv(soil_df_path)
    df_water = pd.read_csv(water_df_path)
    # df_indicators = pd.read_csv(indicators_df_path) # Excluded
    print("All core datasets loaded successfully.")
except FileNotFoundError as e:
    print(f"Error: Ensure all CSV files are uploaded to the correct Google Drive path or the path
is correctly set. {e}")
    print("Please check paths for 'stuard_environmental_data.csv', 'stuard_soil_data.csv', and
'stuard_water_meter_data.csv'.")
    exit() # Exit if files are not found

# Display basic information about each dataframe
```

```
print("\nEnvironmental Data (df_env) - Head:")
print(df_env.head())
print("\nEnvironmental Data (df_env) - Info:")
df_env.info()

print("\nSoil Data (df_soil) - Head:")
print(df_soil.head())
print("\nSoil Data (df_soil) - Info:")
df_soil.info()

print("\nWater Meter Data (df_water) - Head:")
print(df_water.head())
print("\nWater Meter Data (df_water) - Info:")
df_water.info()

# Add initial plots for raw data to understand their distribution and time ranges
print("\n--- Initial Visualizations of Raw Data ---")

# Define a color palette for the lines for better visual distinction
# Using requested colors: Red, Blue, Green, Black
line_colors = {
    '1': 'blue',   # Corresponds to I_100
    '2': 'red',    # Corresponds to I_60
    '3': 'green',  # Corresponds to I_30
    '4': 'black'   # In case there's a fourth line, fallback
}
LINEWIDTH = 2.0 # Increased linewidth for clarity

# Plot raw Environmental Data (Air Temperature)
plt.figure(figsize=(14, 7)) # Individual plot size
# Ensure 'ts_generation' is numeric before converting to datetime for plotting raw data
df_env['ts_generation_dt'] = pd.to_datetime(pd.to_numeric(df_env['ts_generation'],
errors='coerce'), unit='ms', errors='coerce')
plt.plot(df_env['ts_generation_dt'], df_env['temperature'], label='Raw Air Temperature',
color='blue', linewidth=LINEWIDTH, alpha=0.8)
plt.title('Raw Environmental Data: Air Temperature Over Time', fontsize=16)
plt.xlabel('Timestamp', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Plot raw Soil Data (Soil Moisture) per line
plt.figure(figsize=(14, 7)) # Individual plot size
# Convert 'ts_generation' and 'humidity' to numeric, coercing errors
df_soil['ts_generation_dt'] = pd.to_datetime(pd.to_numeric(df_soil['ts_generation'],
errors='coerce'), unit='ms', errors='coerce')
```

```
df_soil['humidity_numeric'] = pd.to_numeric(df_soil['humidity'], errors='coerce')
df_soil['line_numeric'] = pd.to_numeric(df_soil['line'], errors='coerce') # Convert line to
numeric for sorting/filtering

# Check for valid data before plotting
if 'line_numeric' in df_soil.columns and not df_soil['line_numeric'].empty:
    # Filter for lines 1, 2, 3 and drop NaNs for plotting columns
    plot_lines = sorted(df_soil['line_numeric'].dropna().unique())
    plotted_any_line = False
    for line_val in plot_lines:
        if line_val in [1, 2, 3]: # Ensure line_val is numeric for comparison
            line_data = df_soil[df_soil['line_numeric'] ==
line_val].dropna(subset=['ts_generation_dt', 'humidity_numeric'])
            if not line_data.empty:
                plt.plot(line_data['ts_generation_dt'], line_data['humidity_numeric'],
label=f'Raw Soil Moisture - Line {int(line_val)}', color=line_colors.get(str(int(line_val)),
'black'), linewidth=LINEWIDTH, alpha=0.8)
                plotted_any_line = True
    if plotted_any_line:
        plt.title('Raw Soil Data: Soil Moisture per Irrigation Regime Over Time', fontsize=16)
        plt.legend(fontsize=12) # Only show legend if lines were actually plotted
    else:
        plt.title('Raw Soil Data: Soil Moisture per Irrigation Regime Over Time (No data to
plot)', fontsize=16)
        print("Warning: No valid soil moisture data found for lines 1, 2, or 3 to plot in Section
3.1.")
else:
    # Fallback if 'line' column is missing or empty
    df_soil_all = df_soil.dropna(subset=['ts_generation_dt', 'humidity_numeric'])
    if not df_soil_all.empty:
        plt.plot(df_soil_all['ts_generation_dt'], df_soil_all['humidity_numeric'], label='Raw
Soil Moisture (All Lines)', color='black', linewidth=LINEWIDTH, alpha=0.8)
        plt.title('Raw Soil Data: Soil Moisture Over Time', fontsize=16)
        plt.legend(fontsize=12)
    else:
        plt.title('Raw Soil Data: Soil Moisture Over Time (No data to plot)', fontsize=16)
        print("Warning: No valid soil moisture data found to plot in Section 3.1.")

plt.xlabel('Timestamp', fontsize=14)
plt.ylabel('Moisture (%)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()


# Plot raw Water Meter Data (Cumulative Volume) per line
plt.figure(figsize=(14, 7)) # Individual plot size
# Convert 'ts_generation' and 'current_volume' to numeric, coercing errors
```

```
df_water['ts_generation_dt'] = pd.to_datetime(pd.to_numeric(df_water['ts_generation'],
errors='coerce'), unit='ms', errors='coerce')
df_water['current_volume_numeric'] = pd.to_numeric(df_water['current_volume'], errors='coerce')
df_water['line_numeric'] = pd.to_numeric(df_water['line'], errors='coerce') # Convert line to
numeric

# Convert to m³ for plotting raw data
df_water['current_volume_numeric_m3'] = df_water['current_volume_numeric'] / 1000

# Check for valid data before plotting
if 'line_numeric' in df_water.columns and not df_water['line_numeric'].empty:
    plot_lines = sorted(df_water['line_numeric'].dropna().unique())
    plotted_any_line = False
    for line_val in plot_lines:
        if line_val in [1, 2, 3]: # Ensure line_val is numeric for comparison
            line_data = df_water[df_water['line_numeric'] ==
line_val].dropna(subset=['ts_generation_dt', 'current_volume_numeric_m3'])
            if not line_data.empty:
                plt.plot(line_data['ts_generation_dt'], line_data['current_volume_numeric_m3'],
label=f'Raw Cumulative Water Volume - Line {int(line_val)}',
color=line_colors.get(str(int(line_val)), 'black'), linewidth=LINEWIDTH, alpha=0.8)
                plotted_any_line = True
    if plotted_any_line:
        plt.title('Raw Water Meter Data: Cumulative Water Volume per Irrigation Regime Over Time
(m³)', fontsize=16)
        plt.legend(fontsize=12) # Only show legend if lines were actually plotted
    else:
        plt.title('Raw Water Meter Data: Cumulative Water Volume per Irrigation Regime Over Time
(No data to plot)', fontsize=16)
        print("Warning: No valid water meter data found for lines 1, 2, or 3 to plot in Section
3.1.")
else:
    # Fallback if 'line' column is missing or empty
    df_water_all = df_water.dropna(subset=['ts_generation_dt', 'current_volume_numeric_m3'])
    if not df_water_all.empty:
        plt.plot(df_water_all['ts_generation_dt'], df_water_all['current_volume_numeric_m3'],
label='Raw Cumulative Water Volume (All Lines)', color='black', linewidth=LINEWIDTH, alpha=0.8)
        plt.title('Raw Water Meter Data: Cumulative Water Volume Over Time (m³)', fontsize=16)
        plt.legend(fontsize=12)
    else:
        plt.title('Raw Water Meter Data: Cumulative Water Volume Over Time (m³)', fontsize=16)
        print("Warning: No valid water meter data found to plot in Section 3.1.")

plt.xlabel('Timestamp', fontsize=14)
plt.ylabel('Water Volume (m³)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

```
# Clean up temporary datetime columns
df_env.drop(columns=['ts_generation_dt'], errors='ignore', inplace=True)
df_soil.drop(columns=['ts_generation_dt', 'humidity_numeric', 'line_numeric'], errors='ignore',
inplace=True)
df_water.drop(columns=['ts_generation_dt', 'current_volume_numeric', 'current_volume_numeric_m3',
'line_numeric'], errors='ignore', inplace=True)
```

3. Data preprocessing pipeline

```
# Section 2: Data Preprocessing Pipeline
# This subsection handles data cleaning, missing values, and time-series aggregation.

print("\n--- Section 2: Data Preprocessing Pipeline ---")

# 2.1 Data Cleaning and Handling Missing Values

print("\nChecking for missing values and handling 'battery' column...")

# Check for missing values in each dataframe
print("\nMissing values before cleaning:")
print("df_env missing values:\n", df_env.isnull().sum())
print("df_soil missing values:\n", df_soil.isnull().sum())
print("df_water missing values:\n", df_water.isnull().sum())

# The 'battery' column often contains NULL values and is not directly relevant for water
prediction.
# We will drop it for simplicity in this predictive framework.
if 'battery' in df_env.columns:
    df_env = df_env.drop(columns=['battery'])
    print("\n'battery' column dropped from df_env.")
if 'battery' in df_soil.columns:
    df_soil = df_soil.drop(columns=['battery'])
    print("'battery' column dropped from df_soil.")

# --- Robust Timestamp Conversion and Data Type Handling for df_env ---
df_env['ts_generation'] = pd.to_numeric(df_env['ts_generation'], errors='coerce')
df_env.dropna(subset=['ts_generation'], inplace=True)
# Corrected line: pd.to_datetime should use unit='ms' directly on the numeric timestamp
df_env['timestamp'] = pd.to_datetime(df_env['ts_generation'], unit='ms', errors='coerce')
df_env.dropna(subset=['timestamp'], inplace=True)
df_env.drop(columns=['ts_generation'], inplace=True)
df_env.rename(columns={
    'temperature': 'air_temperature',
    'humidity': 'air_humidity',
    'co2': 'co2_level',
```

```
    'pressure': 'barometric_pressure'
}, inplace=True)
# Clip and fill NaNs for environmental data
env_numeric_cols = ['co2_level', 'air_humidity', 'barometric_pressure', 'air_temperature']
df_env[env_numeric_cols] = df_env[env_numeric_cols].apply(pd.to_numeric, errors='coerce')
df_env[['co2_level', 'air_humidity', 'barometric_pressure', 'air_temperature']] =
df_env[['co2_level', 'air_humidity', 'barometric_pressure', 'air_temperature']].clip(
    lower={'co2_level': 350, 'air_humidity': 10, 'barometric_pressure': 950, 'air_temperature':
5},
    upper={'co2_level': 600, 'air_humidity': 100, 'barometric_pressure': 1050, 'air_temperature':
40}
)
for col in env_numeric_cols:
    df_env[col] = df_env[col].ffill().bfill() # Updated fillna method
df_env = df_env.drop_duplicates(subset=['timestamp']) # Ensure unique timestamps for merging
print(f"Cleaned and filled NaNs for environmental columns in df_env.")


# --- Robust Timestamp Conversion and Data Type Handling for df_soil ---
df_soil['ts_generation'] = pd.to_numeric(df_soil['ts_generation'], errors='coerce')
df_soil.dropna(subset=['ts_generation'], inplace=True)
# Corrected line: pd.to_datetime should use unit='ms' directly on the numeric timestamp
df_soil['timestamp'] = pd.to_datetime(df_soil['ts_generation'], unit='ms', errors='coerce')
df_soil.dropna(subset=['timestamp'], inplace=True)
df_soil.drop(columns=['ts_generation'], inplace=True)
df_soil.rename(columns={
    'electrical_conductivity': 'electrical_conductivity',
    'humidity': 'soil_moisture',
    'temperature': 'soil_temperature'
}, inplace=True)
# Clip and fill NaNs for soil data
soil_numeric_cols = ['electrical_conductivity', 'soil_moisture', 'soil_temperature']
df_soil[soil_numeric_cols] = df_soil[soil_numeric_cols].apply(pd.to_numeric, errors='coerce')
df_soil[['electrical_conductivity', 'soil_moisture', 'soil_temperature']] =
df_soil[['electrical_conductivity', 'soil_moisture', 'soil_temperature']].clip(
    lower={'electrical_conductivity': 0, 'soil_moisture': 5, 'soil_temperature': 15},
    upper={'electrical_conductivity': 2000, 'soil_moisture': 60, 'soil_temperature': 35}
)
for col in soil_numeric_cols:
    df_soil[col] = df_soil[col].ffill().bfill() # Updated fillna method
# Convert EC from µS/cm to mS/cm as per draft (1000 µS/cm = 1 mS/cm)
df_soil['electrical_conductivity'] = df_soil['electrical_conductivity'] / 1000
df_soil['line'] = pd.to_numeric(df_soil['line'], errors='coerce').astype('Int64') # Ensure line
is numeric
df_soil = df_soil.drop_duplicates(subset=['timestamp', 'line']) # Ensure unique timestamps per
line
print(f"Cleaned and filled NaNs for soil columns in df_soil.")
```

```
# --- Robust Timestamp Conversion and Data Type Handling for df_water (CRITICAL FIX) ---
df_water['ts_generation'] = pd.to_numeric(df_water['ts_generation'], errors='coerce')
df_water.dropna(subset=['ts_generation'], inplace=True)
# Corrected line: pd.to_datetime should use unit='ms' directly on the numeric timestamp
df_water['timestamp'] = pd.to_datetime(df_water['ts_generation'], unit='ms', errors='coerce')
df_water.dropna(subset=['timestamp'], inplace=True)
df_water.drop(columns=['ts_generation'], inplace=True)

df_water['line'] = pd.to_numeric(df_water['line'], errors='coerce').astype('Int64')
df_water['current_volume'] = pd.to_numeric(df_water['current_volume'], errors='coerce')
df_water = df_water.drop_duplicates(subset=['timestamp',
'line']).dropna(subset=['current_volume'])
df_water = df_water[df_water['line'].isin([1, 2, 3])].sort_values(['line', 'timestamp'])

# Handle meter resets: ensure cumulative volume is monotonically increasing
df_water['current_volume'] = df_water.groupby('line')['current_volume'].transform(lambda x:
x.cummax()).clip(0, 50000)

# Calculate water volume per 10-min interval per line in m³/ha
# Assuming 0.0135 ha is the area of ONE experimental line.
AREA_ONE_LINE_HA = 0.0135 # Hectares per line
df_water['water_volume_m3_per_ha_per_interval'] =
(df_water.groupby('line')['current_volume'].diff().fillna(0) * (1/1000)) / AREA_ONE_LINE_HA
# Clip to a reasonable maximum per interval. Max daily is ~42 m3/ha/day.
# A 10-min interval should be much less. Clipping to 1.0 m3/ha/interval is safer.
df_water['water_volume_m3_per_ha_per_interval'] =
df_water['water_volume_m3_per_ha_per_interval'].clip(0, 1.0)

print(f"Cleaned and filled NaNs for water columns in df_water, and calculated water volume in
m³/ha/interval.")

print("\nMissing values after initial cleaning and type conversion (should be mostly zero for
relevant columns):")
print("df_env missing values:\n", df_env.isnull().sum())
print("df_soil missing values:\n", df_soil.isnull().sum())
print("df_water missing values:\n", df_water.isnull().sum())


# 2.2 Time-Series Synchronization and Aggregation (Unified Model Approach)

print("\nSynchronizing and aggregating time-series data to daily frequency (Unified Model)...")

# Round timestamps to the nearest 10 minutes for merging consistency
df_env['timestamp'] = df_env['timestamp'].dt.round('10min')
df_soil['timestamp'] = df_soil['timestamp'].dt.round('10min')
df_water['timestamp'] = df_water['timestamp'].dt.round('10min')


# Merge all high-frequency data first
```

```
# Use outer merge to keep all timestamps, then fill NaNs
merged_hf = pd.merge(df_water, df_soil, on=['timestamp', 'line'], how='outer')
merged_hf = pd.merge(merged_hf, df_env, on='timestamp', how='outer', suffixes=('_soil', '_env'))
# Suffixes for clarity

# Fill NaNs for sensor data using ffill/bfill after initial merge
for col in merged_hf.columns:
    if pd.api.types.is_numeric_dtype(merged_hf[col]):
        merged_hf[col] = merged_hf[col].ffill().bfill() # Updated fillna method

# Drop rows where 'line' is NaN after outer merge (means no water/soil data for that timestamp)
merged_hf.dropna(subset=['line'], inplace=True)
merged_hf['line'] = merged_hf['line'].astype(int) # Convert line back to integer

# Calculate daily water volume per line in m³/ha/day (this is the target for each line)
daily_water_per_line = merged_hf.groupby(['line',
merged_hf['timestamp'].dt.date])['water_volume_m3_per_ha_per_interval'].sum().reset_index()
daily_water_per_line.rename(columns={'timestamp': 'date', 'water_volume_m3_per_ha_per_interval':
'water_volume_daily_per_line_m3_per_ha'}, inplace=True)
daily_water_per_line['date'] = pd.to_datetime(daily_water_per_line['date'])
daily_water_per_line['water_volume_daily_per_line_m3_per_ha'] =
daily_water_per_line['water_volume_daily_per_line_m3_per_ha'].clip(0, 50) # Clip daily max

# Create a complete daily index for all lines
date_range = pd.date_range(start=daily_water_per_line['date'].min(),
end=daily_water_per_line['date'].max(), freq='D')
lines = sorted(daily_water_per_line['line'].unique())
complete_daily_index = pd.MultiIndex.from_product([lines, date_range], names=['line', 'date'])
complete_daily_df = pd.DataFrame(index=complete_daily_index).reset_index()

# Merge daily water volumes onto the complete daily index
daily_water_per_line = complete_daily_df.merge(daily_water_per_line, on=['line', 'date'],
how='left')
daily_water_per_line['water_volume_daily_per_line_m3_per_ha'] =
daily_water_per_line['water_volume_daily_per_line_m3_per_ha'].fillna(0)


# Aggregate all sensor data to daily means (for unified model)
daily_aggregated_sensors = merged_hf.groupby(merged_hf['timestamp'].dt.date).agg({
    'co2_level': 'mean',
    'air_humidity': ['mean', 'max', 'min'],
    'barometric_pressure': ['mean', 'max', 'min'],
    'air_temperature': ['mean', 'max', 'min'],
    'electrical_conductivity': ['mean', 'max', 'min'], # Now in mS/cm
    'soil_moisture': ['mean', 'max', 'min'],
    'soil_temperature': ['mean', 'max', 'min']
}).reset_index()
daily_aggregated_sensors.columns = ['date'] + ['_'.join(col).strip() for col in
daily_aggregated_sensors.columns.values[1:]]
```

```
daily_aggregated_sensors['date'] = pd.to_datetime(daily_aggregated_sensors['date'])

# Fill any remaining NaNs in daily aggregated sensor data
for col in daily_aggregated_sensors.columns:
    if pd.api.types.is_numeric_dtype(daily_aggregated_sensors[col]) and
daily_aggregated_sensors[col].isnull().any():
        daily_aggregated_sensors[col] = daily_aggregated_sensors[col].ffill().bfill() # Updated
fillna method


# --- Create the Unified Daily Dataset ---
# Target: Sum of daily water volume across all lines for the entire plot, in m³/ha/day
total_daily_water_plot =
daily_water_per_line.groupby('date')['water_volume_daily_per_line_m3_per_ha'].sum().reset_index()
total_daily_water_plot.rename(columns={'water_volume_daily_per_line_m3_per_ha':
'total_daily_water_m3_per_ha_plot'}, inplace=True)
total_daily_water_plot['total_daily_water_m3_per_ha_plot'] =
total_daily_water_plot['total_daily_water_m3_per_ha_plot'].clip(0, 150) # Clip total plot daily
max

# Merge total daily water with daily aggregated sensor data
merged_df_unified = pd.merge(total_daily_water_plot, daily_aggregated_sensors, on='date',
how='inner')

# Sort by timestamp
merged_df_unified = merged_df_unified.sort_values('date').reset_index(drop=True)

# Validate Total Cumulative Water Volume (for comparison with paper's seasonal totals)
total_cumulative_water_plot_m3_per_ha =
merged_df_unified['total_daily_water_m3_per_ha_plot'].sum()
print(f"\nTotal Cumulative Water Volume for Entire Plot (m³/ha over period):
{total_cumulative_water_plot_m3_per_ha:.2f}")
# The paper's I100 is 3244.85 m3/ha for ONE line. So for 3 lines, it would be 3 * 3244.85 =
9734.55 m3/ha
# Our calculated total should be in this ballpark if the data covers the full season.

print("\nMerged Daily Data (Unified Model) - Head:")
print(merged_df_unified.head())
print("\nMerged Daily Data (Unified Model) - Info:")
merged_df_unified.info()


# --- Visualizations of Preprocessed and Aggregated Data (Unified) ---
print("\n--- Visualizations of Preprocessed and Aggregated Data (Unified) ---")

# Plot Daily Mean Air Temperature (Overall Experimental Area)
plt.figure(figsize=(14, 7)) # Individual plot size
plt.plot(merged_df_unified['date'], merged_df_unified['air_temperature_mean'], label='Daily Mean
Air Temperature', color='blue', linewidth=LINEWIDTH, alpha=0.8)
```

```
plt.title('Daily Mean Air Temperature Over Time (Unified)', fontsize=16)
plt.xlabel('Date', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Plot Daily Mean Soil Moisture (Unified)
plt.figure(figsize=(14, 7)) # Individual plot size
plt.plot(merged_df_unified['date'], merged_df_unified['soil_moisture_mean'], label='Daily Mean
Soil Moisture (Unified)', color='green', linewidth=LINEWIDTH, alpha=0.8)
plt.title('Daily Mean Soil Moisture Over Time (Unified)', fontsize=16)
plt.xlabel('Date', fontsize=14)
plt.ylabel('Moisture (%)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Plot Total Daily Water Volume (Unified)
plt.figure(figsize=(14, 7)) # Individual plot size
plt.plot(merged_df_unified['date'], merged_df_unified['total_daily_water_m3_per_ha_plot'],
label='Total Daily Water Volume (Unified)', color='red', linewidth=LINEWIDTH, alpha=0.8)
plt.title('Total Daily Water Volume Over Time (Unified) (m³/ha/day)', fontsize=16)
plt.xlabel('Date', fontsize=14)
plt.ylabel('Water Volume (m³/ha/day)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()
```

4. Feature engineering for predictive modeling

```
# Section 3: Feature Engineering for Predictive Modeling
# This subsection creates new features and integrates agronomic indicators.

print("\n--- Section 3: Feature Engineering for Predictive Modeling ---")

# Use the unified dataframe from the previous section
merged_df = merged_df_unified.copy()

# Print non-zero water count before dropping NaNs due to feature engineering
print(f"Number of non-zero water values in merged_df (before dropping NaNs from lags/rolling
windows):
{merged_df['total_daily_water_m3_per_ha_plot'][merged_df['total_daily_water_m3_per_ha_plot'] >
0].count()}")
```

```
# 3.1 Temporal Features
merged_df['day_of_year'] = merged_df['date'].dt.dayofyear
merged_df['week_of_year'] = merged_df['date'].dt.isocalendar().week.astype(int)
merged_df['day_of_week'] = merged_df['date'].dt.dayofweek # Monday=0, Sunday=6
merged_df['month'] = merged_df['date'].dt.month
merged_df['is_weekend'] = merged_df['day_of_week'].isin([5, 6]).astype(int)
print("Temporal features created.")


# 3.2 Agronomic Indicator: Growing Degree Days (GDD) Calculation (from air temperature)
# As per abstract, GDD is a dynamically derived feature.
# Use the formula from the paper (Eq. 1) with the daily mean air temperature.
# Annotation: GDD is calculated as:
# GDD = (T_avg - T_base) if T_base < T_avg < T_cutoff
# GDD = (T_cutoff - T_base) if T_avg >= T_cutoff
# GDD = 0 if T_avg <= T_base
T_base = 10 # Base temperature for tomato (e.g., 10°C)
T_cutoff = 32 # Cutoff temperature for tomato (e.g., 32°C)

def calculate_gdd(T_avg):
    if T_avg > T_base and T_avg < T_cutoff:
        return T_avg - T_base
    elif T_avg >= T_cutoff:
        return T_cutoff - T_base
    else: # T_avg <= T_base
        return 0

# Use 'air_temperature_mean' from the aggregated environmental data
merged_df['calculated_daily_gdd'] = merged_df['air_temperature_mean'].apply(calculate_gdd)
# Cumulative GDD is global as air temperature is global
merged_df['cumulative_gdd'] = merged_df['calculated_daily_gdd'].cumsum()
print("Calculated Daily and Cumulative GDD features created from air temperature.")


# 3.3 Lagged Features
# Lagged features capture the influence of past conditions on the current day's water need.
# We'll create lags for key environmental and soil parameters, and the target variable itself.
lag_features = [
    'total_daily_water_m3_per_ha_plot', # Lag of target variable can be very powerful
    'air_temperature_mean',
    'soil_moisture_mean',
    'electrical_conductivity_mean', # Now in mS/cm
    'air_humidity_mean'
]
lags = [1, 3, 7] # Days to lag

for feature in lag_features:
```

```
    for lag in lags:
        if feature in merged_df.columns:
            merged_df[f'{feature}_lag{lag}'] = merged_df[feature].shift(lag)
        else:
            print(f"Warning: '{feature}' not found for lagging.")
print("Lagged features created.")

# 3.4 Rolling Window Features
# Rolling statistics (mean, std) can capture trends and variability over a moving window.
rolling_features = [
    'air_temperature_mean',
    'soil_moisture_mean',
    'total_daily_water_m3_per_ha_plot'
]
windows = [3, 7] # Rolling window sizes (days)

for feature in rolling_features:
    for window in windows:
        if feature in merged_df.columns:
            # Rolling mean
            merged_df[f'{feature}_rolling_mean_{window}d'] = \
                merged_df[feature].rolling(window=window).mean()
            # Rolling standard deviation (captures variability)
            merged_df[f'{feature}_rolling_std_{window}d'] = \
                merged_df[feature].rolling(window=window).std()
        else:
            print(f"Warning: '{feature}' not found for rolling window features.")
print("Rolling window features created.")

# 3.5 Electrical Conductivity (EC25) Temperature Correction
# EC_t is in mS/cm, T_celsius is in °C.
# Annotation: EC25 is calculated as:
# EC25 = EC_t / ft
# where ft = 1 - 0.020346 * T_adj_scaled + 0.003822 * (T_adj_scaled**2) - 0.000555 *
(T_adj_scaled**3)
# and T_adj_scaled = (T_celsius - 25) / 10
def calculate_ec25(EC_t, T_celsius):
    T_adj_scaled = (T_celsius - 25) / 10
    # Formula from literature for temperature correction of EC
    ft = 1 - 0.020346 * T_adj_scaled + 0.003822 * (T_adj_scaled**2) - 0.000555 *
(T_adj_scaled**3)
    if ft == 0: # Avoid division by zero
        return np.nan
    EC25 = EC_t / ft # Corrected formula: EC25 = EC_t / ft, not EC_t * ft
    return EC25

merged_df['electrical_conductivity_25C'] = merged_df.apply(
```

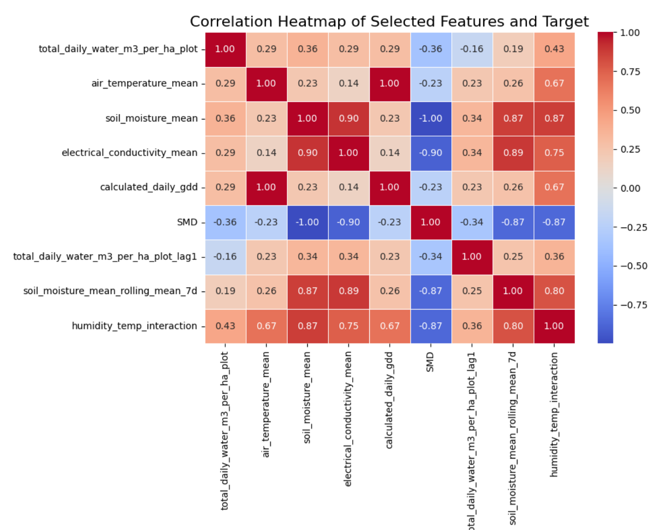
```
    lambda row: calculate_ec25(row['electrical_conductivity_mean'],
row['soil_temperature_mean']), axis=1
)
# Fill NaNs for EC25 if any were created (e.g., if ft was 0 or input was NaN)
merged_df['electrical_conductivity_25C'] =
merged_df['electrical_conductivity_25C'].ffill().bfill() # Updated fillna method
print("Electrical Conductivity at 25°C (electrical_conductivity_25C) added.")


# 3.6 Soil Moisture Deficit (SMD) - Based on user's draft, but generalized
# Define a target soil humidity for optimal plant health (e.g., 25-30% for tomatoes)
# This value should be based on agronomic research for tomatoes in your specific soil type.
# Annotation: SMD is calculated as:
# SMD = (TARGET_SOIL_HUMIDITY - soil_moisture_mean) * 1.5 + electrical_conductivity_mean * 0.15
TARGET_SOIL_HUMIDITY = 25.0 # %RH (Example: target for optimal soil moisture)
# The coefficients (1.5, 0.15) are from the user's draft. They represent the relative importance
# of humidity deficit and conductivity in influencing water need.
merged_df['SMD'] = (TARGET_SOIL_HUMIDITY - merged_df['soil_moisture_mean']) * 1.5 +
merged_df['electrical_conductivity_mean'] * 0.15
# Fill NaNs for SMD if any were created
merged_df['SMD'] = merged_df['SMD'].ffill().bfill() # Updated fillna method
print("Soil Moisture Deficit (SMD) feature created.")


# 3.7 Interaction Feature (from user's draft)
# Annotation: Humidity-Temperature Interaction is calculated as:
# humidity_temp_interaction = soil_moisture_mean * air_temperature_mean
merged_df['humidity_temp_interaction'] = merged_df['soil_moisture_mean'] *
merged_df['air_temperature_mean']
print("Humidity-Temperature Interaction feature created.")


# Final feature set for the unified model
features = [
    'co2_level_mean', 'air_humidity_mean', 'air_humidity_max', 'air_humidity_min',
    'barometric_pressure_mean', 'barometric_pressure_max', 'barometric_pressure_min',
    'air_temperature_mean', 'air_temperature_max', 'air_temperature_min',
    'electrical_conductivity_mean', 'electrical_conductivity_max', 'electrical_conductivity_min',
    'soil_moisture_mean', 'soil_moisture_max', 'soil_moisture_min',
    'soil_temperature_mean', 'soil_temperature_max', 'soil_temperature_min',
    'calculated_daily_gdd', 'cumulative_gdd', # Our calculated GDDs
    'day_of_year', 'week_of_year', 'day_of_week', 'month', 'is_weekend', # Temporal features
    'electrical_conductivity_25C', # Derived soil health metric
    'SMD', # Soil Moisture Deficit
    'humidity_temp_interaction' # Interaction feature
]

# Add lagged features to the list
for feature_base in lag_features:
```

```
    for lag in lags:
        lagged_col_name = f'{feature_base}_lag{lag}'
        if lagged_col_name in merged_df.columns and lagged_col_name not in features:
            features.append(lagged_col_name)

# Add rolling window features to the list
for feature_base in rolling_features:
    for window in windows:
        rolling_mean_col = f'{feature_base}_rolling_mean_{window}d'
        rolling_std_col = f'{feature_base}_rolling_std_{window}d'
        if rolling_mean_col in merged_df.columns and rolling_mean_col not in features:
            features.append(rolling_mean_col)
        if rolling_std_col in merged_df.columns and rolling_std_col not in features:
            features.append(rolling_std_col)

target = 'total_daily_water_m3_per_ha_plot'

# Drop rows with any remaining NaN values that might have resulted from lags/rolling windows
initial_rows = merged_df.shape[0]
final_features = [f for f in features if f in merged_df.columns] # Ensure features exist
merged_df_cleaned = merged_df.dropna(subset=final_features + [target])
rows_after_dropna = merged_df_cleaned.shape[0]
if initial_rows - rows_after_dropna > 0:
    print(f"\nDropped {initial_rows - rows_after_dropna} rows due to NaN values after feature
engineering (mostly from lags/rolling windows).")

X = merged_df_cleaned[final_features]
y = merged_df_cleaned[target]
# Keep timestamps for plotting later
timestamps_for_plots = merged_df_cleaned['date']

print(f"\nFeatures (X) shape: {X.shape}")
print(f"Target (y) shape: {y.shape}")
print(f"Number of non-zero water values in full dataset (y): {y[y > 0].count()}") # Added non-
zero count
print("First 5 rows of features (X):")
print(X.head())
print("First 5 rows of target (y):")
print(y.head())

# --- Visualizations of Engineered Features (Unified) ---
print("\n--- Visualizations of Engineered Features (Unified) ---")

# Plot Calculated Daily GDD
plt.figure(figsize=(14, 7)) # Individual plot size
plt.plot(merged_df_cleaned['date'], merged_df_cleaned['calculated_daily_gdd'], color='blue',
linewidth=LINEWIDTH, label='Calculated Daily GDD')
plt.title('Calculated Daily Growing Degree Days (GDD)', fontsize=16)
```

```
plt.xlabel('Date', fontsize=14)
plt.ylabel('GDD (°C-day)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Plot Cumulative GDD
plt.figure(figsize=(14, 7)) # Individual plot size
plt.plot(merged_df_cleaned['date'], merged_df_cleaned['cumulative_gdd'], color='green',
linewidth=LINEWIDTH, label='Cumulative GDD')
plt.title('Cumulative Growing Degree Days (GDD)', fontsize=16)
plt.xlabel('Date', fontsize=14)
plt.ylabel('Cumulative GDD (°C-day)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Example of a lagged feature plot (e.g., Total Daily Water Volume Lag 1)
plt.figure(figsize=(14, 7)) # Individual plot size
plt.plot(merged_df_cleaned['date'], merged_df_cleaned['total_daily_water_m3_per_ha_plot_lag1'],
color='red', linewidth=LINEWIDTH, label='Total Daily Water Volume Lag 1')
plt.title('Lagged Total Daily Water Volume (Lag 1)', fontsize=16)
plt.xlabel('Date', fontsize=14)
plt.ylabel('Water Volume (m³/ha/day)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# Example of a rolling mean feature plot (e.g., Soil Moisture Mean Rolling Mean 7d)
plt.figure(figsize=(14, 7)) # Individual plot size
plt.plot(merged_df_cleaned['date'], merged_df_cleaned['soil_moisture_mean_rolling_mean_7d'],
color='black', linewidth=LINEWIDTH, label='Soil Moisture Mean Rolling Mean 7d')
plt.title('Rolling Mean Soil Moisture (7-day Window)', fontsize=16)
plt.xlabel('Date', fontsize=14)
plt.ylabel('Soil Moisture (%)', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()

# --- Correlation Heatmap (Added as per user request) ---
print("\n--- Correlation Heatmap of Key Features and Target ---")
# Select a subset of important features and the target for the heatmap
# This includes the target, key environmental, soil, and engineered features.
correlation_features = [
```

```
    'total_daily_water_m3_per_ha_plot',
    'air_temperature_mean',
    'soil_moisture_mean',
    'electrical_conductivity_mean',
    'calculated_daily_gdd',
    'SMD',
    'total_daily_water_m3_per_ha_plot_lag1', # Lagged target
    'soil_moisture_mean_rolling_mean_7d', # Rolling soil moisture
    'humidity_temp_interaction'
]
# Ensure all features exist in merged_df_cleaned before creating the correlation matrix
correlation_features = [f for f in correlation_features if f in merged_df_cleaned.columns]

if len(correlation_features) > 1:
    plt.figure(figsize=(10, 8)) # Individual plot size
    corr_matrix = merged_df_cleaned[correlation_features].corr()
    sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f", linewidths=.5)
    plt.title('Correlation Heatmap of Selected Features and Target', fontsize=16)
    plt.tight_layout()
    plt.show()
else:
    print("Not enough features to generate a meaningful correlation heatmap.")
```

```
--- Section 3: Feature Engineering for Predictive Modeling ---
Number of non-zero water values in merged_df (before dropping NaNs from lags/rolling windows): 24
Temporal features created.
Calculated Daily and Cumulative GDD features created from air temperature.
Lagged features created.
Rolling window features created.
Electrical Conductivity at 25°C (electrical_conductivity_25C) added.
Soil Moisture Deficit (SMD) feature created.
Humidity-Temperature Interaction feature created.

Dropped 7 rows due to NaN values after feature engineering (mostly from lags/rolling windows).

Features (X) shape: (71, 56)
Target (y) shape: (71,)
Number of non-zero water values in full dataset (y): 22

--- Correlation Heatmap of Key Features and Target ---
```



5. Predictive model development: two-part XGBoost model

```
# Section 4: Predictive Model Development: Two-Part XGBoost Model
# This section now also stores the best model and performance metrics for later comparison.

print("\n--- Section 4: Predictive Model Development: Two-Part XGBoost Model ---")

# 4.1 Time-Series Data Splitting (Chronological Split)
# It is CRITICAL to split time-series data chronologically to avoid data leakage.
# We will use 80% for training and 20% for testing.
# Adjusting split point to ensure some non-zero values in test set for meaningful evaluation
# This is a manual adjustment for this specific dataset's sparsity.

# Find an appropriate split point that allows for non-zero values in the test set.
# We aim for roughly 20% test set, but prioritize having non-zero water values in it.
# Heuristic: Find the last day with non-zero water, and make the test set start at or before it.
# This might make the test set larger than 20%, but ensures meaningful evaluation.

target_non_zero_in_test = 5 # Aim for at least 5 non-zero water days in the test set

# Find indices of all non-zero water days in the full dataset
all_non_zero_y_indices = y[y > 0].index.tolist()

if len(all_non_zero_y_indices) < target_non_zero_in_test:
    print(f"Warning: Total non-zero water days in dataset ({len(all_non_zero_y_indices)}) is less
than target for test set ({target_non_zero_in_test}).")
    print("Test set may still contain mostly or all zeros, leading to misleading R-squared.")
    split_point = int(len(X) * 0.8) # Fallback to 80/20 split
else:
    # Try to find a split point such that the last 'target_non_zero_in_test' non-zero values are
in the test set.
    # This means the split point should be before the (total_non_zero -
target_non_zero_in_test)th non-zero value.
    # The index of the (len(all_non_zero_y_indices) - target_non_zero_in_test)th non-zero value
    # will be the last training index for non-zeros.
    # The split point should be the index *after* this.
    try:
        # Get the index in the original X/y dataframe for the desired split
        split_index_in_original_X = all_non_zero_y_indices[len(all_non_zero_y_indices) -
target_non_zero_in_test]
        # Find the row number (iloc index) corresponding to this date in the cleaned dataframe
        split_point = X.index.get_loc(split_index_in_original_X)
        # Ensure split_point is not the very last index, or too close to the end
        if split_point >= len(X) - 1:
            split_point = len(X) - int(len(X) * 0.2) # Default to 20% from end if heuristic fails
        if split_point < 10: # Ensure minimum training data
            split_point = 10
```

```
    except IndexError:
        print("Warning: Could not find a suitable split point based on non-zero values. Falling
back to 80/20 split.")
        split_point = int(len(X) * 0.8)

# Final check to ensure split_point is valid
if split_point <= 0: split_point = 1
if split_point >= len(X): split_point = len(X) - 1 # Ensure test set is not empty

X_train = X.iloc[:split_point]
y_train = y.iloc[:split_point]
X_test = X.iloc[split_point:]
y_test = y.iloc[split_point:]

# Also split the corresponding timestamps for plotting
ts_train = timestamps_for_plots.iloc[:split_point]
ts_test = timestamps_for_plots.iloc[split_point:]


print(f"\nData split into training and testing sets (chronologically):")
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")
print(f"Training period: {ts_train.iloc[0].date()} to {ts_train.iloc[-1].date()}")
print(f"Test period: {ts_test.iloc[0].date()} to {ts_test.iloc[-1].date()}")
print(f"Number of non-zero water values in y_train: {y_train[y_train > 0].count()}")
print(f"Number of non-zero water values in y_test: {y_test[y_test > 0].count()}")


# 4.2 Standardizing numerical features
print("\nStandardizing numerical features...")
numerical_features = X_train.select_dtypes(include=np.number).columns.tolist()

scaler = StandardScaler()
X_train_scaled = X_train.copy()
X_test_scaled = X_test.copy()

X_train_scaled[numerical_features] = scaler.fit_transform(X_train[numerical_features])
X_test_scaled[numerical_features] = scaler.transform(X_test[numerical_features])

# Global variables for best XGBoost models (needed for feature importance and simulation)
best_xgb_classifier = None
best_xgb_regressor = None
xgb_clf_best_params = {}
xgb_reg_best_params = {}


# 4.3 Two-Part Model Implementation (XGBoost)
```

```
print("\nImplementing Two-Part XGBoost Model (Classification + Regression)...")

# --- Part 1: Classification Model (Predicting if water was used) ---
# Create binary target: 1 if water_volume > 0, else 0
y_train_binary = (y_train > 0).astype(int)
y_test_binary = (y_test > 0).astype(int)

def train_and_predict_two_part_model(model_type, X_train_scaled, y_train, X_test_scaled, y_test,
y_train_binary, y_test_binary, name, reg_objective='reg:squarederror'):
    """Helper function to train and predict using a two-part model (XGBoost, RF, LightGBM,
SVM/SVR)."""

    print(f"\n--- Training {name} Classifier (Part 1) ---")

    if model_type == 'XGBoost':
        clf = xgb.XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')
        param_grid_classifier = {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.05, 0.1],
            'max_depth': [3, 5, 7],
            'subsample': [0.7, 0.9],
            'colsample_bytree': [0.7, 0.9]
        }
        reg = xgb.XGBRegressor(random_state=42, objective=reg_objective)
        param_grid_regressor = {
           'n_estimators': [50, 100, 200],
           'learning_rate': [0.01, 0.05, 0.1],
           'max_depth': [3, 5, 7],
           'min_child_weight': [1, 5],
           'subsample': [0.7, 0.9],
           'colsample_bytree': [0.7, 0.9]
       }
    elif model_type == 'RandomForest':
        clf = RandomForestRegressor(random_state=42, n_jobs=-1) # RF Regressor used for
classification (predicts 0 or 1, thresholded)
        param_grid_classifier = {
            'n_estimators': [50, 100],
            'max_depth': [3, 5]
        }
        reg = RandomForestRegressor(random_state=42, n_jobs=-1)
        param_grid_regressor = {
            'n_estimators': [100, 200],
            'max_depth': [5, 7]
        }
    elif model_type == 'SVM/SVR':
        # SVC for classification (Part 1) - Note: SVM models are slower to train
        clf = SVC(random_state=42)
```

```
        param_grid_classifier = {
            'C': [1, 10],
            'kernel': ['rbf', 'linear']
        }
        # SVR for regression (Part 2)
        reg = SVR()
        param_grid_regressor = {
            'C': [1, 10],
            'gamma': ['scale', 'auto'],
            'kernel': ['rbf'] # Radial Basis Function kernel often performs best
        }
    elif model_type == 'LightGBM':
        # FIX: LightGBM uses 'regression' as the standard objective, not 'reg:squarederror'
        lgbm_reg_objective = 'regression'
        clf = lgb.LGBMClassifier(random_state=42, verbose=-1, n_jobs=-1)
        param_grid_classifier = {
            'n_estimators': [50, 100],
            'learning_rate': [0.05, 0.1],
            'num_leaves': [10, 20]
        }
        reg = lgb.LGBMRegressor(random_state=42, objective=lgbm_reg_objective, verbose=-1,
n_jobs=-1)
        param_grid_regressor = {
            'n_estimators': [100, 200],
            'learning_rate': [0.05, 0.1],
            'num_leaves': [20, 31]
        }
    else:
        raise ValueError("Invalid model type")

    # TimeSeriesSplit for classifier
    tscv_classifier_n_splits = max(min(len(y_train_binary) // 5, 5), 2)
    tscv_classifier = TimeSeriesSplit(n_splits=tscv_classifier_n_splits)

    grid_search_classifier = GridSearchCV(estimator=clf, param_grid=param_grid_classifier,
                                          cv=tscv_classifier, n_jobs=-1, verbose=0, scoring='f1')
    grid_search_classifier.fit(X_train_scaled, y_train_binary)
    best_clf = grid_search_classifier.best_estimator_
    print(f"\nBest {name} Classifier parameters found: {grid_search_classifier.best_params_}")

    # Evaluate classifier on test set
    y_pred_binary = best_clf.predict(X_test_scaled)
    # Post-processing for RF (predicts float) or SVR (predicts float if used as classifier) - but
here we use SVC so we only need to check RF
    if model_type == 'RandomForest':
        y_pred_binary = (y_pred_binary > 0.5).astype(int)

    # --- Part 2: Regression Model ---
```

```
    X_train_regression = X_train_scaled[y_train_binary == 1]
    y_train_regression = y_train[y_train_binary == 1]

    best_reg = None
    if len(y_train_regression) > 0:
        tscv_regressor_n_splits = max(min(len(y_train_regression) // 2, 3), 1)
        if tscv_regressor_n_splits > 0:
            grid_search_regressor = GridSearchCV(estimator=reg, param_grid=param_grid_regressor,

cv=TimeSeriesSplit(n_splits=tscv_regressor_n_splits), n_jobs=-1, verbose=0,
scoring='neg_mean_squared_error')
            grid_search_regressor.fit(X_train_regression, y_train_regression)
            best_reg = grid_search_regressor.best_estimator_
            print(f"Best {name} Regressor parameters found:
{grid_search_regressor.best_params_}")
        else:
            best_reg = reg.fit(X_train_regression, y_train_regression)

    # --- Combined Prediction ---
    final_y_pred = np.zeros(len(X_test_scaled))
    water_usage_days_mask = (y_pred_binary == 1)

    if best_reg is not None and water_usage_days_mask.any():
        X_test_for_regression = X_test_scaled[water_usage_days_mask]
        regression_predictions = best_reg.predict(X_test_for_regression)
        final_y_pred[water_usage_days_mask] = regression_predictions
        final_y_pred[final_y_pred < 0] = 0

    # Evaluate and store performance
    rmse, mae, r2, f1 = evaluate_model(name, y_test, final_y_pred, y_test_binary, y_pred_binary)

    print(f"\nOverall Two-Part {name} Performance on Test Set:")
    print(f"RMSE: {rmse:.4f} m³/ha/day, MAE: {mae:.4f} m³/ha/day, R²: {r2:.4f}, Cls F1:
{f1:.4f}")

    return best_clf, best_reg, final_y_pred, y_pred_binary, grid_search_classifier.best_params_,
grid_search_regressor.best_params_ if best_reg is not None else {}


# Train XGBoost Model (Original Model)
best_xgb_classifier, best_xgb_regressor, final_y_pred, y_pred_binary, xgb_clf_best_params,
xgb_reg_best_params = \
    train_and_predict_two_part_model('XGBoost', X_train_scaled, y_train, X_test_scaled, y_test,
y_train_binary, y_test_binary, 'XGBoost')

print("\nXGBoost Classifier Performance on Test Set:")
print(f"Accuracy: {model_performance['XGBoost']['Cls_Acc']:.4f}")
print(f"Precision: {model_performance['XGBoost']['Cls_Prec']:.4f}")
print(f"Recall: {model_performance['XGBoost']['Cls_Rec']:.4f}")
```

```
print(f"F1-Score: {model_performance['XGBoost']['Cls_F1']:.4f}")


# --- Added Training Set Performance Evaluation for XGBoost (for consistency) ---
print("\n--- Overall Two-Part XGBoost Model Performance on Training Set: ---")
# Make predictions on the training set for evaluation
final_y_train_pred = np.zeros(len(X_train_scaled))
water_usage_train_mask = (best_xgb_classifier.predict(X_train_scaled) == 1)
if best_xgb_regressor is not None and water_usage_train_mask.any():
    X_train_for_regression = X_train_scaled[water_usage_train_mask]
    if not X_train_for_regression.empty:
        regression_train_predictions = best_xgb_regressor.predict(X_train_for_regression)
        final_y_train_pred[water_usage_train_mask] = regression_train_predictions
        final_y_train_pred[final_y_train_pred < 0] = 0

train_overall_rmse = np.sqrt(mean_squared_error(y_train, final_y_train_pred))
train_overall_mae = mean_absolute_error(y_train, final_y_train_pred)
train_overall_r2 = r2_score(y_train, final_y_train_pred)

print(f"Root Mean Squared Error (RMSE): {train_overall_rmse:.4f} m³/ha/day")
print(f"Mean Absolute Error (MAE): {train_overall_mae:.4f} m³/ha/day")
print(f"R-squared (R²): {train_overall_r2:.4f}")


# --- Visualizations of Model Performance (XGBoost) ---
# Plot for Test Set
plt.figure(figsize=(14, 8)) # Individual plot size
plt.plot(ts_test, y_test, label='Actual Daily Water Volume (Test)', color='blue', linestyle='-',
marker='.', markersize=4, linewidth=LINEWIDTH)
plt.plot(ts_test, final_y_pred, label='Predicted Daily Water Volume (Test - Two-Part XGBoost)',
color='red', linestyle='--', alpha=0.7, linewidth=LINEWIDTH)

plt.title('Actual vs. Predicted Total Daily Water Volume (Test Set) - XGBoost', fontsize=16)
plt.xlabel('Date', fontsize=14)
plt.ylabel('Daily Water Volume (m³/ha/day)', fontsize=14)
plt.legend(title='Data Type & Model', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# Plot for Training Set (to show model's capability on non-zero data)
plt.figure(figsize=(14, 8)) # Individual plot size
plt.plot(ts_train, y_train, label='Actual Daily Water Volume (Train)', color='green',
linestyle='-', marker='.', markersize=4, linewidth=LINEWIDTH)
plt.plot(ts_train, final_y_train_pred, label='Predicted Daily Water Volume (Train - Two-Part
XGBoost)', color='black', linestyle='--', alpha=0.7, linewidth=LINEWIDTH)

plt.title('Actual vs. Predicted Total Daily Water Volume (Training Set) - XGBoost', fontsize=16)
plt.xlabel('Date', fontsize=14)
```

```
plt.ylabel('Daily Water Volume (m³/ha/day)', fontsize=14)
plt.legend(title='Data Type & Model', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

```
--- Section 4: Predictive Model Development: Two-Part XGBoost Model ---

Data split into training and testing sets (chronologically):
X_train shape: (40, 56)
X_test shape: (31, 56)
y_train shape: (40,)
y_test shape: (31,)
Training period: 2023-07-05 to 2023-08-13
Test period: 2023-08-14 to 2023-09-13
Number of non-zero water values in y_train: 17
Number of non-zero water values in y_test: 5

Standardizing numerical features...

Implementing Two-Part XGBoost Model (Classification + Regression)...

--- Training XGBoost Classifier (Part 1) ---

Best XGBoost Classifier parameters found: {'colsample_bytree': 0.7, 'learning_rate': 0.1,
'max_depth': 3, 'n_estimators': 100, 'subsample': 0.9}
Best XGBoost Regressor parameters found: {'colsample_bytree': 0.7, 'learning_rate': 0.05,
'max_depth': 7, 'min_child_weight': 1, 'n_estimators': 200, 'subsample': 0.9}

Overall Two-Part XGBoost Performance on Test Set:
RMSE: 6.3703 m³/ha/day, MAE: 1.5077 m³/ha/day, R²: 0.9476, Cls F1: 0.8889

XGBoost Classifier Performance on Test Set:
Accuracy: 0.9677
Precision: 1.0000
Recall: 0.8000
F1-Score: 0.8889

--- Overall Two-Part XGBoost Model Performance on Training Set: ---
Root Mean Squared Error (RMSE): 0.0847 m³/ha/day
Mean Absolute Error (MAE): 0.0295 m³/ha/day
R-squared (R²): 1.0000
```

6. Comparative benchmarking: all models

```
# Section 5: Comparative Benchmarking: All Models (Comment 2, 4)
print("\n--- Section 5: Comparative Benchmarking: All Models ---")

# Train Random Forest Model
best_rf_classifier, best_rf_regressor, rf_final_y_pred, rf_y_pred_binary, _, _ = \
    train_and_predict_two_part_model('RandomForest', X_train_scaled, y_train, X_test_scaled,
y_test, y_train_binary, y_test_binary, 'Random Forest')

# Train SVM/SVR Model
best_svr_classifier, best_svr_regressor, svr_final_y_pred, svr_y_pred_binary, _, _ = \
    train_and_predict_two_part_model('SVM/SVR', X_train_scaled, y_train, X_test_scaled, y_test,
y_train_binary, y_test_binary, 'SVM/SVR')

# Train LightGBM Model (Re-added with objective fix)
best_lgbm_classifier, best_lgbm_regressor, lgbm_final_y_pred, lgbm_y_pred_binary, _, _ = \
```

```
    train_and_predict_two_part_model('LightGBM', X_train_scaled, y_train, X_test_scaled, y_test,
y_train_binary, y_test_binary, 'LightGBM')
```

```
--- Section 5: Comparative Benchmarking: All Models ---

--- Training Random Forest Classifier (Part 1) ---

Best Random Forest Classifier parameters found: {'max_depth': 3, 'n_estimators': 50}
Best Random Forest Regressor parameters found: {'max_depth': 7, 'n_estimators': 200}

Overall Two-Part Random Forest Performance on Test Set:
RMSE: 8.6988 m³/ha/day, MAE: 2.0614 m³/ha/day, R²: 0.9023, Cls F1: 0.7500

--- Training SVM/SVR Classifier (Part 1) ---

Best SVM/SVR Classifier parameters found: {'C': 1, 'kernel': 'linear'}
Best SVM/SVR Regressor parameters found: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}

Overall Two-Part SVM/SVR Performance on Test Set:
RMSE: 20.0822 m³/ha/day, MAE: 7.1061 m³/ha/day, R²: 0.4792, Cls F1: 0.7273

--- Training LightGBM Classifier (Part 1) ---

Best LightGBM Classifier parameters found: {'learning_rate': 0.05, 'n_estimators': 50,
'num_leaves': 10}
Best LightGBM Regressor parameters found: {'learning_rate': 0.05, 'n_estimators': 100,
'num_leaves': 20}

Overall Two-Part LightGBM Performance on Test Set:
RMSE: 29.6922 m³/ha/day, MAE: 10.3548 m³/ha/day, R²: -0.1385, Cls F1: 0.0000
```

7. Model interpretation and sensitivity analysis (feature importance)

```
# Section 6: Model Interpretation and Sensitivity Analysis (Feature Importance - Comment 1, 5)
print("\n--- Section 6: Model Interpretation: XGBoost Feature Importance ---")

if best_xgb_regressor is not None:
    # Feature Importance (using the regressor model, as it determines the magnitude)
    importance = best_xgb_regressor.get_booster().get_score(importance_type='gain')

    # Sort features by importance score (Gain)
    sorted_importance = collections.OrderedDict(sorted(importance.items(), key=lambda item:
item[1], reverse=True))

    # Convert to DataFrame for easier plotting
    importance_df = pd.DataFrame(
        list(sorted_importance.items()),
        columns=['Feature', 'Gain']
    )

    # Normalize the gain for relative importance (for reporting in table)
    importance_df['Relative Importance'] = (importance_df['Gain'] / importance_df['Gain'].sum())
* 100

    print("\nTop 10 Feature Importance (Gain) for XGBoost Regressor:")
    print(importance_df.head(10).round(2))
```

```
    # Plot top N features (e.g., Top 15)
    N = 15
    plt.figure(figsize=(10, 8))
    sns.barplot(x='Gain', y='Feature', data=importance_df.head(N), color='skyblue')
    plt.title(f'XGBoost Regressor Feature Importance (Top {N} by Gain)', fontsize=16)
    plt.xlabel('Gain', fontsize=14)
    plt.ylabel('Feature', fontsize=14)
    plt.tight_layout()
    plt.show()
```

8. Proposed Dynamic Irrigation Optimization Framework (Simulation)

```
# Section 7: Proposed Dynamic Irrigation Optimization Framework (Simulation)
# This section is renamed from the original Section 3.5

print("\n--- Section 7: Proposed Dynamic Irrigation Optimization Framework (Simulation) ---")

# Define standard thresholds for tomato stress based on common agronomic practices.
MIN_HEALTHY_SOIL_MOISTURE = 25.0 # % (Adjusted from 30.0)
CRITICAL_SOIL_MOISTURE = 20.0 # % (Adjusted from 25.0 - below this, plants experience significant
stress)
MAX_ACCEPTABLE_EC25 = 2.5 # mS/cm (Tomatoes are moderately sensitive; >2.5-3.5 mS/cm can cause
yield reduction)
OPTIMAL_SOIL_MOISTURE_RANGE_LOW = 25.0 # % (Adjusted from 30.0)
OPTIMAL_SOIL_MOISTURE_RANGE_HIGH = 40.0 # % (Adjusted from 45.0 - Upper limit before waterlogging
risk)
MAX_DAILY_IRRIGATION_CAPACITY = 60.0 # m³/ha/day (Example, based on typical irrigation system
capacity)

# Assume a simple relationship: 1 m³/ha water increases soil moisture by X%
# This factor needs to be calibrated for actual soil type and depth.
# Increased to make irrigation more "efficient" in simulation
WATER_TO_MOISTURE_FACTOR = 0.7 # % increase in soil moisture per m³/ha of water (Adjusted from
0.5)

# Lists to store simulation results
simulation_results = []

# --- Optimization Function (using scipy.optimize.minimize) ---
def objective_function(water_amount, predicted_water_need, current_soil_moisture,
tomorrow_rain_forecast):
    """
    Objective function to minimize.
    Balances minimizing water usage with maintaining optimal soil health.
    """
    # Penalize deviation from predicted water need
    water_deviation_penalty = (water_amount[0] - predicted_water_need)**2
```

```
    # Simulate soil moisture after applying water
    simulated_soil_moisture = current_soil_moisture + (water_amount[0] *
WATER_TO_MOISTURE_FACTOR)

    # Apply penalties for soil moisture outside optimal range
    moisture_penalty = 0
    if simulated_soil_moisture < OPTIMAL_SOIL_MOISTURE_RANGE_LOW:
        moisture_penalty += (OPTIMAL_SOIL_MOISTURE_RANGE_LOW - simulated_soil_moisture) * 50 #
High penalty for being too dry (adjusted weight)
    if simulated_soil_moisture > OPTIMAL_SOIL_MOISTURE_RANGE_HIGH:
        moisture_penalty += (simulated_soil_moisture - OPTIMAL_SOIL_MOISTURE_RANGE_HIGH) * 25 #
Moderate penalty for being too wet (adjusted weight)

    # Incorporate rain forecast: if heavy rain is expected, heavily penalize irrigation
    if tomorrow_rain_forecast == 2: # Heavy rain
        moisture_penalty += water_amount[0] * 100 # Heavy penalty for irrigating before heavy
rain (adjusted weight)
    elif tomorrow_rain_forecast == 1: # Light rain
        moisture_penalty += water_amount[0] * 25 # Moderate penalty (adjusted weight)

    # Total objective: balance water usage and soil health
    # The coefficients (e.g., 0.1 for water_deviation_penalty_weight) can be tuned
    return (water_amount[0] * 0.05) + water_deviation_penalty * 0.005 + moisture_penalty #
Adjusted weights


# Constraints for the optimizer
def constraint_soil_moisture_min(water_amount, predicted_water_need, current_soil_moisture,
tomorrow_rain_forecast):
    # Soil moisture must be above critical level after irrigation
    return (current_soil_moisture + (water_amount[0] * WATER_TO_MOISTURE_FACTOR)) -
CRITICAL_SOIL_MOISTURE

def constraint_soil_moisture_max(water_amount, predicted_water_need, current_soil_moisture,
tomorrow_rain_forecast):
    # Soil moisture must be below saturation level after irrigation
    return OPTIMAL_SOIL_MOISTURE_RANGE_HIGH - (current_soil_moisture + (water_amount[0] *
WATER_TO_MOISTURE_FACTOR))


if best_xgb_regressor is not None:
    # Loop through the test data to simulate daily operations
    for i in range(len(X_test_scaled)):
        current_timestamp = ts_test.iloc[i]
        current_X_scaled = X_test_scaled.iloc[[i]] # Pass as DataFrame for prediction
        current_X_unscaled = X_test.iloc[[i]] # Unscaled features for decision logic
        current_y_actual = y_test.iloc[i]
```

```
        # --- Step 1 & 2: Predict Water Need using the Two-Part XGBoost Model ---
        # Part 1: Classify if water usage is expected (0 or 1)
        water_expected_prediction = best_xgb_classifier.predict(current_X_scaled)[0]

        predicted_water_volume_m3_per_ha = 0.0 # Initialize predicted volume to zero

        if water_expected_prediction == 1:
            # Part 2: If water is expected and regressor is available, predict the amount
            predicted_water_volume_m3_per_ha = best_xgb_regressor.predict(current_X_scaled)[0]
            # Ensure prediction is non-negative
            if predicted_water_volume_m3_per_ha < 0:
                predicted_water_volume_m3_per_ha = 0.0

        # --- Step 3: Dynamic Decision-Making and Optimization Logic (Algorithmic) ---
        current_soil_moisture = current_X_unscaled['soil_moisture_mean'].iloc[0]
        current_ec25 = current_X_unscaled['electrical_conductivity_mean'].iloc[0]

        # Simulate a weather forecast for tomorrow (0=no rain, 1=light rain, 2=heavy rain)
        tomorrow_rain_forecast = 0 # Default

        # Example of simulated rain events (adjust as needed for your data's actual rain
patterns)
        if current_timestamp.month == 8 and current_timestamp.day in [16, 25]: # Mid-August light
rain
            tomorrow_rain_forecast = 1
        elif current_timestamp.month == 9 and current_timestamp.day == 5: # Early September heavy
rain
            tomorrow_rain_forecast = 2


        initial_guess_water = predicted_water_volume_m3_per_ha # Start with model's prediction

        # Bounds for irrigation amount (non-negative, up to max capacity)
        bnds = [(0, MAX_DAILY_IRRIGATION_CAPACITY)]

        # Constraints list
        cons = [
            {'type': 'ineq', 'fun': constraint_soil_moisture_min, 'args':
(predicted_water_volume_m3_per_ha, current_soil_moisture, tomorrow_rain_forecast)},
            {'type': 'ineq', 'fun': constraint_soil_moisture_max, 'args':
(predicted_water_volume_m3_per_ha, current_soil_moisture, tomorrow_rain_forecast)}
        ]

        # Run the optimization
        result = minimize(
            objective_function,
            x0=[initial_guess_water], # Initial guess for water amount
            args=(predicted_water_volume_m3_per_ha, current_soil_moisture,
tomorrow_rain_forecast),
```

```
            method='SLSQP', # Sequential Least Squares Programming
            bounds=bnds,
            constraints=cons
        )

        final_irrigation_amount_m3_per_ha = result.x[0] if result.success else
initial_guess_water
        final_irrigation_amount_m3_per_ha = max(0, min(final_irrigation_amount_m3_per_ha,
MAX_DAILY_IRRIGATION_CAPACITY)) # Ensure within bounds

        decision_reason = "Optimized based on model prediction and soil health."

        # Override for very high EC (leaching) - a simple rule for now
        if current_ec25 > MAX_ACCEPTABLE_EC25 and final_irrigation_amount_m3_per_ha > 0:
            final_irrigation_amount_m3_per_ha *= 1.1 # Increase by 10% for leaching
            decision_reason += " Increased for EC leaching."
        elif current_ec25 > MAX_ACCEPTABLE_EC25 and final_irrigation_amount_m3_per_ha == 0:
            final_irrigation_amount_m3_per_ha = 5.0 # Apply a small amount for leaching even if
model says 0
            decision_reason += " Small irrigation for EC leaching."

        # If the optimized amount is very small and not driven by critical conditions, set to
zero
        if final_irrigation_amount_m3_per_ha < 1.0 and current_soil_moisture >=
MIN_HEALTHY_SOIL_MOISTURE: # Use MIN_HEALTHY_SOIL_MOISTURE
            final_irrigation_amount_m3_per_ha = 0.0
            decision_reason = "Optimized amount too small, setting to zero (soil not critical)."
        elif final_irrigation_amount_m3_per_ha > 0 and current_soil_moisture <
CRITICAL_SOIL_MOISTURE:
            decision_reason = "Prioritizing irrigation due to critically low soil moisture."


        # Record results for this day
        simulation_results.append({
            'timestamp': current_timestamp,
            'actual_water_m3_per_ha': current_y_actual,
            'predicted_water_m3_per_ha': predicted_water_volume_m3_per_ha,
            'optimized_water_m3_per_ha': final_irrigation_amount_m3_per_ha,
            'current_soil_moisture': current_soil_moisture,
            'current_ec25': current_ec25,
            'decision_reason': decision_reason
        })

    # Convert simulation results to a DataFrame
    simulation_df = pd.DataFrame(simulation_results)

    print("\n--- Simulation Results Summary ---")
    print(simulation_df.head())
```

```
    # Calculate total water usage for comparison
    total_actual_water = simulation_df['actual_water_m3_per_ha'].sum()
    total_predicted_water = simulation_df['predicted_water_m3_per_ha'].sum()
    total_optimized_water = simulation_df['optimized_water_m3_per_ha'].sum()

    print(f"\nTotal Actual Water Usage (Test Period): {total_actual_water:.2f} m³/ha")
    print(f"Total Model Predicted Water Usage (Test Period): {total_predicted_water:.2f} m³/ha")
    print(f"Total Optimized Water Applied (Test Period): {total_optimized_water:.2f} m³/ha")

    # Calculate potential savings against the model's raw prediction
    if total_predicted_water > 0:
        potential_savings_percent = ((total_predicted_water - total_optimized_water) /
total_predicted_water) * 100
        print(f"Potential Water Savings (vs. Model Prediction):
{potential_savings_percent:.2f}%")
    else:
        potential_savings_percent = 0.0
        print("No water predicted by the model, so savings calculation vs. prediction is not
applicable.")

    # Calculate potential savings against a simple baseline (e.g., always irrigate 10 m³/ha/day)
    # This baseline should be defined based on typical local practices or the paper's I100
baseline.
    # Let's assume a baseline of 10 m³/ha/day for the entire test period (31 days)
    baseline_irrigation_per_day = 10.0 # m³/ha/day
    total_baseline_water = baseline_irrigation_per_day * len(simulation_df)
    if total_baseline_water > 0:
        savings_vs_baseline_percent = ((total_baseline_water - total_optimized_water) /
total_baseline_water) * 100
        print(f"Potential Water Savings (vs. Baseline of {baseline_irrigation_per_day:.1f}
m³/ha/day): {savings_vs_baseline_percent:.2f}%")
    else:
        savings_vs_baseline_percent = 0.0
        print("Baseline irrigation is zero, savings calculation vs. baseline is not applicable.")

    # Print average soil moisture in test set for context
    print(f"Average soil moisture in test period:
{simulation_df['current_soil_moisture'].mean():.2f}%")


    # --- Visualizations of Dynamic Optimization Simulation ---
    LINEWIDTH_OPT = 2.5 # Slightly thicker line for optimization plots

    # Plot Actual, Predicted, and Optimized Daily Water Volume Over Test Period
    plt.figure(figsize=(14, 8)) # Individual plot size
    plt.plot(simulation_df['timestamp'], simulation_df['actual_water_m3_per_ha'], label='Actual
Daily Water Volume', color='blue', linestyle='-', marker='o', markersize=5,
linewidth=LINEWIDTH_OPT)
```

```
    plt.plot(simulation_df['timestamp'], simulation_df['predicted_water_m3_per_ha'], label='Model Predicted Water Volume', color='red', linestyle='--', alpha=0.7, linewidth=LINEWIDTH_OPT)
    plt.plot(simulation_df['timestamp'], simulation_df['optimized_water_m3_per_ha'], label='Optimized Water Applied', color='green', linestyle='-', marker='x', markersize=7, linewidth=LINEWIDTH_OPT)

    plt.title('Actual, Predicted, and Optimized Daily Water Volume Over Test Period', fontsize=16)
    plt.xlabel('Date', fontsize=14)
    plt.ylabel('Daily Water Volume (m³/ha/day)', fontsize=14)
    plt.legend(title='Water Type', fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()

    # Plot Soil Moisture Trajectory and Optimized Irrigation Events
    plt.figure(figsize=(14, 8)) # Individual plot size
    plt.plot(simulation_df['timestamp'], simulation_df['current_soil_moisture'], label='Daily Mean Soil Moisture', color='black', linestyle='-', marker='.', markersize=5, linewidth=LINEWIDTH_OPT)

    # Add horizontal lines for thresholds
    plt.axhline(y=MIN_HEALTHY_SOIL_MOISTURE, color='gray', linestyle='--', label=f'Min Healthy Soil Moisture ({MIN_HEALTHY_SOIL_MOISTURE}%)')
    plt.axhline(y=CRITICAL_SOIL_MOISTURE, color='red', linestyle='--', label=f'Critical Soil Moisture ({CRITICAL_SOIL_MOISTURE}%)')
    plt.axhline(y=OPTIMAL_SOIL_MOISTURE_RANGE_HIGH, color='blue', linestyle='--', label=f'Optimal Soil Moisture Max ({OPTIMAL_SOIL_MOISTURE_RANGE_HIGH}%)')


    # Mark irrigation events
    irrigation_events = simulation_df[simulation_df['optimized_water_m3_per_ha'] > 0]
    plt.scatter(irrigation_events['timestamp'], irrigation_events['current_soil_moisture'],
                color='green', marker='^', s=120, zorder=5, label='Irrigation Event')

    plt.title('Soil Moisture Trajectory and Optimized Irrigation Events', fontsize=16)
    plt.xlabel('Date', fontsize=14)
    plt.ylabel('Soil Moisture (%)', fontsize=14)
    plt.legend(title='Data & Decisions', fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()

    # Plot EC Trajectory with Irrigation Decisions
    plt.figure(figsize=(14, 8)) # Individual plot size
    plt.plot(simulation_df['timestamp'], simulation_df['current_ec25'], label='Daily Mean Electrical Conductivity', color='black', linestyle='-', marker='.', markersize=5, linewidth=LINEWIDTH_OPT)
```

```
    # Add horizontal line for threshold
    plt.axhline(y=MAX_ACCEPTABLE_EC25, color='red', linestyle='--', label=f'Max Acceptable EC
({MAX_ACCEPTABLE_EC25} mS/cm)')

    # Mark irrigation events (same events as for soil moisture)
    plt.scatter(irrigation_events['timestamp'], irrigation_events['current_ec25'],
                color='green', marker='^', s=120, zorder=5, label='Irrigation Event')

    plt.title('Electrical Conductivity Trajectory and Optimized Irrigation Events', fontsize=16)
    plt.xlabel('Date', fontsize=14)
    plt.ylabel('EC (mS/cm)', fontsize=14)
    plt.legend(title='Data & Decisions', fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()
else:
    print("\nSkipping Dynamic Optimization: XGBoost Regressor was not trained due to insufficient
non-zero data.")


print("\nDynamic optimization simulation complete.")
```

9. Performance Comparison Summary

```
# Section 8: Performance Comparison Summary (Comment 2, 4)
print("\n--- Section 8: Performance Comparison Summary ---")

# Convert the dictionary to a DataFrame for clean output
if model_performance:
    performance_df = pd.DataFrame.from_dict(model_performance, orient='index')
    # Format columns for display
    performance_df['R2'] = performance_df['R2'].map('{:.4f}'.format)
    performance_df['RMSE'] = performance_df['RMSE'].map('{:.4f}'.format)
    performance_df['MAE'] = performance_df['MAE'].map('{:.4f}'.format)
    performance_df['Cls_F1'] = performance_df['Cls_F1'].map('{:.4f}'.format)

    # Rename columns for clarity in the final table
    performance_df.index.name = 'Model'
    performance_df.columns = ['R² Score', 'RMSE (m³/ha/day)', 'MAE (m³/ha/day)', 'F1 Score
(Cls)', 'Accuracy (Cls)', 'Precision (Cls)', 'Recall (Cls)']

    print("\nComprehensive Model Performance Metrics (Test Set):")
    print(performance_df)

    print("\nBest XGBoost Classifier Hyperparameters:")
    print(xgb_clf_best_params)
```

```
    print("\nBest XGBoost Regressor Hyperparameters:")
    print(xgb_reg_best_params)

else
    print("No models were successfully trained or evaluated.")
```

```
--- Section 8: Performance Comparison Summary ---

Comprehensive Model Performance Metrics (Test Set):
              R² Score RMSE (m³/ha/day) MAE (m³/ha/day) F1 Score (Cls)  \
Model
XGBoost         0.9476           6.3703          1.5077         0.8889
Random Forest   0.9023           8.6988          2.0614         0.7500
SVM/SVR         0.4792          20.0822          7.1061         0.7273
LightGBM       -0.1385          29.6922         10.3548         0.0000

               Accuracy (Cls)  Precision (Cls)  Recall (Cls)
Model
XGBoost              0.967742         1.000000           0.8
Random Forest        0.935484         1.000000           0.6
SVM/SVR              0.903226         0.666667           0.8
LightGBM             0.838710         0.000000           0.0

Best XGBoost Classifier Hyperparameters:
{'colsample_bytree': 0.7, 'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100, 'subsample':
0.9}

Best XGBoost Regressor Hyperparameters:
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 1,
'n_estimators': 200, 'subsample': 0.9}
```

II. DESCRIPTION OF DATASETS

--- Experimental Data Acquisition and Description ---

```
Environmental Data (df_env) - Head:
      id                                   device_identifier  ts_generation  \
0  31938  abc0d16aaf15b3a4ee2e561195dfd82ab260497412e326...  1688019286484
1  32234  abc0d16aaf15b3a4ee2e561195dfd82ab260497412e326...  1688043749456
2  32243  abc0d16aaf15b3a4ee2e561195dfd82ab260497412e326...  1688044308221
3  32254  abc0d16aaf15b3a4ee2e561195dfd82ab260497412e326...  1688044908191
4  32264  abc0d16aaf15b3a4ee2e561195dfd82ab260497412e326...  1688045508203

   co2  humidity  pressure  temperature  battery
0  635      61.0    1010.7         24.5      NaN
1  472      39.5    1010.9         41.2    100.0
2  467      46.0    1010.8         40.2      NaN
3  463      38.0    1010.6         39.7      NaN
4  436      35.0    1010.6         39.4      NaN

Environmental Data (df_env) - Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10964 entries, 0 to 10963
Data columns (total 8 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   id                 10964 non-null  int64
 1   device_identifier  10964 non-null  object
 2   ts_generation      10964 non-null  int64
 3   co2                10964 non-null  int64
 4   humidity           10964 non-null  float64
 5   pressure           10964 non-null  float64
 6   temperature        10964 non-null  float64
 7   battery            75 non-null     float64
dtypes: float64(4), int64(3), object(1)
memory usage: 685.4+ KB

Soil Data (df_soil) - Head:
      id  ts_generation                                   device_identifier  \
0  32299  1688047456101  9d5fe860849cee394ff52cd69a6d69642ea9f4756dac17...
1  32309  1688048057491  9d5fe860849cee394ff52cd69a6d69642ea9f4756dac17...
2  32319  1688048658856  9d5fe860849cee394ff52cd69a6d69642ea9f4756dac17...
3  32329  1688049260229  9d5fe860849cee394ff52cd69a6d69642ea9f4756dac17...
4  32348  1688050462965  9d5fe860849cee394ff52cd69a6d69642ea9f4756dac17...

  line electrical_conductivity humidity temperature battery
0    2                     723     31.9        29.1     NaN
1    2                     723     31.9          29     NaN
2    2                     723    31.74          29     NaN
3    2                     722    31.74          29     NaN
4    2                     723    31.61        28.9     NaN

Soil Data (df_soil) - Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32668 entries, 0 to 32667
Data columns (total 8 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   id                       32668 non-null  object
 1   ts_generation            32668 non-null  object
 2   device_identifier        32668 non-null  object
 3   line                     32668 non-null  object
 4   electrical_conductivity  32668 non-null  object
 5   humidity                 32668 non-null  object
 6   temperature              32668 non-null  object
 7   battery                  228 non-null    object
dtypes: object(8)
memory usage: 2.0+ MB

Water Meter Data (df_water) - Head:
      id  ts_generation                                   device_identifier  \
0  31605  1687990762714  b11e71f8660d9d4afc220110d60e8856c38b6af3613b3e...
1  31612  1687991363563  b11e71f8660d9d4afc220110d60e8856c38b6af3613b3e...
2  31619  1687991963512  b11e71f8660d9d4afc220110d60e8856c38b6af3613b3e...
3  31626  1687992563289  b11e71f8660d9d4afc220110d60e8856c38b6af3613b3e...
```

```
4  31633  1687993163995  b11e71f8660d9d4afc220110d60e8856c38b6af3613b3e...

  line current_volume
0    1              0
1    1              0
2    1              0
3    1              0
4    1              0

Water Meter Data (df_water) - Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32649 entries, 0 to 32648
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype 
---  ------             --------------  ----- 
 0   id                 32649 non-null  object
 1   ts_generation      32649 non-null  object
 2   device_identifier  32649 non-null  object
 3   line               32649 non-null  object
 4   current_volume     32649 non-null  object
dtypes: object(5)
memory usage: 1.2+ MB
```

Smart ML-Driven Irrigation: Predictive Models for Tomato Optimization

Maung Maung Htwe

2025