DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2024

DeepCM Based Sensor Calibration and Data Based Sensor Fusion

<Jike Li>

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Jike Li <jkli02441@gmail.com>
Information and Network Engineering
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
Some place

Examiner

The Professor Place
KTH Royal Institute of Technology

Supervisor

The Supervisor
Place
KTH Royal Institute of Technology

Abstract

This thesis addresses significant challenges in calibrating low-cost CO_2 sensors, predicting short-term CO_2 levels, and integrating sensor data for enhanced accuracy in environmental monitoring. Low-cost sensors often face issues such as accuracy drift, sparse data collection, and restricted spatial coverage, limiting their effectiveness. To overcome these challenges, several advanced methods were employed.

For sensor calibration, the thesis introduces the Deep Calibration Method (DeepCM), utilizing deep learning to model complex, non-linear sensor behaviors. Within this framework, the Sliding Window Supervised Calibration method achieved the best results, with a Root Mean Square Error (RMSE) of 4.56, significantly enhancing calibration accuracy compared to other methods.

In the short-term prediction of CO_2 levels, various Long Short-Term Memory (LSTM) models were employed, including the Single-sensor LSTM Model, Single-sensor LSTM Encoder-Decoder Model, and Multi-sensor LSTM Encoder-Decoder Model. The Single-sensor LSTM Model performed notably well, achieving an RMSE of 2.19 and a Mean Absolute Error (MAE) of 1.64, demonstrating superior prediction accuracy over traditional time-series models.

For sensor fusion, Gaussian Process Regression (GPR) was applied to integrate data from multiple sensors. The study explored both time-dimension GPR Sensor Fusion and space-dimension GPR Sensor Fusion models. Specifically, the Time Dimension GPR model exhibited an RMSE of 35, which, while slightly higher than the RMSE of 32 achieved by the Weighted Average model, provided superior handling of data volatility and noise, leading to more reliable long-term predictions.

Chapter 1 introduces the project background, outlining the problem, goals, and methodologies. Chapter 2 provides an extensive review of existing methods in outlier detection, sensor calibration, short-term prediction, and sensor fusion, setting the stage for the approaches developed in this thesis. Chapter 3 details the methodology, explaining the construction, training, and application of the DeepCM, various LSTM models, and GPR models. In Chapter 4, the implementation of these methods is discussed, followed by Chapter 5, which presents and analyzes the results, quantifying the improvements achieved by each model. Finally, Chapter 6 concludes the thesis with a summary of findings, an evaluation of limitations, and recommendations for future research.

Keywords

CO_2 Sensor Calibration, Deep Learning, Long Short-Term Memory (LSTM), Deep Calibration Method (DeepCM), Sensor Fusion, Gaussian Process Regression (GPR)

Abstract

Denna
avhandling behandlar betydande utmaningar i kalibrering av lågkostnadssensorer för koldioxid (CO_2), korttidsprognostisering av CO_2-nivåer och integrering av sensordata för förbättrad noggrannhet i miljöövervakning. Lågkostnadssensorer stöter ofta på problem såsom noggrannhetsdrift, sparsam datainsamling och begränsad spatial täckning, vilket begränsar deras effektivitet. För att övervinna dessa utmaningar tillämpades flera avancerade metoder.

För sensorernas kalibrering introducerar avhandlingen Deep Calibration Method (DeepCM), som använder djupinlärning för att modellera komplexa, icke-linjära sensorbeteenden. Inom denna ramverk uppnådde Sliding Window Supervised Calibration-metoden de bästa resultaten, med ett Root Mean Square Error (RMSE) på 4.56, vilket signifikant förbättrade kalibreringsnoggrannheten jämfört med andra metoder.

När det gäller korttidsprognostisering av CO_2-nivåer användes olika Long Short-Term Memory (LSTM) modeller, inklusive Single-sensor LSTM Model, Single-sensor LSTM Encoder-Decoder Model och Multi-sensor LSTM Encoder-Decoder Model. Single-sensor LSTM Model presterade särskilt väl, med ett RMSE på 2.19 och ett Mean Absolute Error (MAE) på 1.64, vilket visade överlägsen prognosnoggrannhet jämfört med traditionella tidsseriemodeller.

För sensorfusion tillämpades Gaussian Process Regression (GPR) för att integrera data från flera sensorer. Studien utforskade både Time Dimension GPR Sensor Fusion och Space Dimension GPR Sensor Fusion modeller. Specifikt visade Time Dimension GPR-modellen ett RMSE på 35, vilket, även om det var något högre än RMSE på 32 som uppnåddes av Weighted Average-modellen, erbjöd överlägsen hantering av datavolatilitet och brus, vilket ledde till mer tillförlitliga långsiktiga prognoser.

Kapitel 1 introducerar projektets bakgrund, skisserar problematiken, målen och metoderna. Kapitel 2 ger en omfattande översyn av befintliga metoder inom avvikelseupptäckning, sensor kalibrering, korttidsprognostisering och sensorfusion, och sätter scenen för de tillvägagångssätt som utvecklats i denna avhandling. Kapitel 3 beskriver metodiken, förklarar konstruktionen, träningen och tillämpningen av DeepCM, olika LSTM-modeller och GPR-modeller. I Kapitel 4 diskuteras implementeringen av dessa metoder, följt av Kapitel 5, som presenterar och analyserar resultaten, och kvantifierar de förbättringar som uppnåtts med varje modell. Slutligen sammanfattar Kapitel 6 avhandlingen med en sammanfattning av fynden, en

utvärdering av begränsningar och rekommendationer för framtida forskning.

Nyckelord

CO_2 Sensor Kalibrering, Djupinlärning, Long Short-Term Memory (LSTM), Djup Kalibreringsmetod (DeepCM), Sensorfusion, Gaussisk Processregression (GPR)

Acknowledgements

Write a short acknowledgements. Don't forget to give some credit to the examiner and supervisor.

Acronyms

Contents

Chapter 1

Introduction

This chapter aims to provide the reader with some background, the report's purpose and objective. The limitations and the relevance of the project is also discussed in this chapter.

1.1 Background

In today's era of increasing reliance on environmental monitoring, the demand for affordable CO_2 sensors has grown substantially. These sensors play a crucial role in ensuring indoor air quality, energy efficiency, and regulatory compliance in various sectors. However, one of the primary challenges with low-cost CO_2 sensors is maintaining their accuracy through effective calibration.[1]

Calibration is essential to align sensor readings with known reference standards, thereby ensuring reliable data output. For low-cost CO_2 sensors, achieving accurate calibration is particularly critical due to potential variations and drift in sensor performance over time.[2] Effective calibration methods not only enhance the accuracy of CO_2 measurements but also enable informed decision-making in building management systems, smart cities, and industrial processes.[3]

In addition to calibration, predicting short-term variations in CO_2 levels is essential for optimizing ventilation strategies and improving indoor air quality. [4]Advanced statistical techniques and machine learning algorithms can analyze historical sensor data to forecast CO_2 concentrations, enabling proactive adjustments to environmental conditions.

Moreover, sensor fusion techniques, which integrate data from multiple low-cost sensors, offer a promising solution to enhance monitoring accuracy and coverage. By combining measurements from different sensor types or locations, sensor fusion mitigates individual sensor limitations and provides a comprehensive view of environmental conditions.[5]

As the field of low-cost CO_2 sensors continues to evolve, effective calibration methods, accurate short-term prediction models, and advanced sensor fusion techniques

will play pivotal roles in maximizing sensor reliability and utility across diverse applications.

1.2 Problem

In the realm of CO_2 monitoring, the use of low-cost sensors poses several significant challenges:

1. Challenges in Integrating Low-Cost CO_2 Sensors

Low-cost CO_2 sensors exhibit lower accuracy compared to high-cost counterparts. Achieving and maintaining accurate calibration poses a challenge, as these sensors may drift over time or exhibit inconsistent performance.[6, 7] Calibration is crucial to align sensor readings with reference standards, but using all high-cost sensor data for calibration may not be practical due to cost constraints and logistical challenges. Selectively integrating a small subset of high-cost sensor data for calibration is also challenging, given the disparity in sensor capabilities. This complicates the calibration process and limits effective use, as the combined dataset needs careful validation and alignment to ensure meaningful results.

2. Low Sampling Frequency and Limited Data Volume

Low-cost CO_2 sensors typically operate at lower sampling frequencies and produce fewer data points compared to high-cost sensors. This limitation results in sparse data coverage, making it difficult to capture rapid fluctuations or subtle changes in CO_2 levels. As a consequence, short-term trends may be missed, and the ability to analyze and predict long-term dependencies is compromised. The sparse data also impacts the accuracy and reliability of monitoring systems, as there may be gaps in the data that prevent a comprehensive understanding of environmental conditions.

3. Suboptimal Performance in Gas Monitoring Optimization

Inherent limitations in low-cost sensors contribute to suboptimal performance in optimizing CO_2 monitoring systems.[8]These sensors may have reduced sensitivity or response times, leading to delays in detecting changes in CO_2 levels. This affects the ability to achieve high precision and reliability in real-time monitoring applications, where timely and accurate data is crucial for effective decision-making. Improving the performance of low-cost sensors in gas monitoring optimization is essential to enhance the overall effectiveness and utility of monitoring systems.

4. Restricted Spatial Coverage

Low-cost CO_2 sensors often cover smaller geographic areas compared to high-cost counterparts. This limitation restricts their utility in applications requiring comprehensive spatial coverage across large regions or complex environments. In scenarios where monitoring large indoor spaces, outdoor environments, or complex

industrial settings is necessary, the limited spatial coverage of low-cost sensors may not provide sufficient data granularity or spatial resolution. This can hinder the ability to monitor and manage CO_2 levels effectively across diverse and expansive areas.

Addressing these challenges is critical for advancing the reliability, accuracy, and scalability of CO_2 monitoring technologies. In environments where precision and real-time responsiveness are paramount, overcoming these limitations will be essential to leveraging the full potential of CO_2 sensor technology.

1.3 Purpose

The purpose of this thesis is to address the challenges and explore methodologies related to low-cost sensor calibration, short-term forecasting of calibrated data, and integrated forecasting of calibrated data. Specifically, the thesis aims to:

1. Investigate calibration methods for enhancing the accuracy of low-cost sensors in environmental monitoring applications.
2. Develop techniques for short-term forecasting using calibrated sensor data to predict near-future trends in CO_2 levels.
3. Explore approaches for integrating calibrated sensor data from diverse sources to improve the accuracy and reliability of long-term CO_2 level predictions.
4. Evaluate the robustness and scalability of calibration methods across different environmental conditions and sensor types.
5. Investigate the impact of various forecasting models on the accuracy and reliability of short-term CO_2 level predictions.
6. Assess the effectiveness of data fusion techniques in enhancing the predictive capabilities of integrated sensor data for long-term CO_2 level forecasts.

By focusing on these objectives, the thesis aims to contribute to the advancement of CO_2 monitoring technologies, particularly in environments where precision and real-time responsiveness are critical.

1.4 Goal

The goals of this thesis are to achieve the following objectives related to low-cost sensor calibration, short-term forecasting, and integrated sensor fusion:

1. Establish a calibration model capable of operating independently or using only a small subset of high-cost sensor data.
2. Develop an effective short-term forecasting model for predicting near-future trends in CO_2 levels based on calibrated sensor data.
3. Create a sensor fusion forecasting model that integrates data across time and space dimensions to enhance long-term CO_2 level predictions.

4. Compare the performance of the established models with existing calibration and forecasting methodologies through rigorous evaluation metrics.
5. Assess the robustness and scalability of the calibration model across different environmental conditions and sensor configurations.
6. Evaluate the accuracy and reliability of the short-term forecasting model against alternative prediction approaches in different scenarios.
7. Validate the effectiveness of the sensor fusion model in improving predictive capabilities compared to single-sensor models across diverse environmental settings.

By accomplishing these goals, the thesis aims to advance the state-of-the-art in CO_2 monitoring technologies, contributing to enhanced precision and efficiency in environmental monitoring applications.

1.5 Methodology

The methodology employed in this thesis integrates theoretical frameworks and advanced techniques for addressing challenges in sensor calibration, short-term forecasting, and sensor fusion in CO_2 monitoring applications.

1. Sensor Calibration

Sensor calibration is crucial for improving the accuracy of low-cost sensors in environmental monitoring. Traditional methods such as linear regression and polynomial regression are effective but limited in handling complex non-linear relationships that often characterize sensor data. To address this limitation, a deep calibration method (DeepCM), is adopted.[9] DeepCM excels in capturing intricate patterns and non-linear dependencies in data, thereby enhancing the accuracy of sensor calibration by learning from large datasets and adapting to diverse sensor characteristics.

2. Short-term Forecasting

Accurate short-term forecasting of CO_2 levels is essential for real-time decision-making in environmental management. While traditional models like ARIMA and exponential smoothing are widely used, they may struggle with capturing long-term dependencies in time-series data. Long Short-Term Memory (LSTM), a type of recurrent neural network (RNN), is chosen for its ability to retain information over extended periods, making it suitable for predicting short-term fluctuations in CO_2 levels.[10] LSTM models have demonstrated superior performance in time-series forecasting tasks due to their capacity to capture temporal dynamics and sequence dependencies effectively.

3. Sensor Fusion

Integrating data from multiple sensors across time and space dimensions is essential for comprehensive CO_2 level predictions. Traditional methods such as Kalman filtering and Bayesian filtering are commonly used for sensor fusion but may struggle with

handling complex noise characteristics and uncertainty in heterogeneous sensor data. Gaussian Process Regression (GPR) offers a robust alternative by modeling uncertainty explicitly and providing a flexible framework for integrating sensor data. GPR has been successfully applied in various domains to combine information from multiple sources, yielding accurate predictions by effectively managing noise and uncertainty in data.

For this degree project, the following methods are selected based on their theoretical foundations and applicability:

- **DeepCM for Sensor Calibration**
  Leveraging deep learning to model non-linear sensor data relationships and enhance calibration accuracy.
- **LSTM for Short-term Forecasting**
  Utilizing recurrent neural networks to capture temporal dependencies and predict near-future CO_2 levels accurately.
- **Gaussian Process Regression for Sensor Fusion**
  Employing probabilistic modeling to integrate heterogeneous sensor data and improve long-term CO_2 level predictions.

This thesis adopts advanced methodologies—DeepCM for calibration, LSTM for short-term forecasting, and GPR for sensor fusion—to advance CO_2 monitoring capabilities. These methods leverage theoretical insights and computational power to enhance accuracy, reliability, and predictive capabilities in environmental sensing applications.

1.6 Delimitations

This study is delimited by several constraints that may influence the interpretation and generalizability of the results:

1. **Sensor Data Quality and Reliability:** The accuracy and reliability of the calibrated sensor data heavily depend on the quality of the input data and the calibration process itself. Variations in sensor performance, environmental conditions, and calibration procedures may introduce uncertainties into the results.
2. **Temporal and Spatial Coverage:** The sensor network's temporal and spatial coverage may be limited, potentially leading to gaps in data collection. Variations in sensor deployment density and placement across different locations could affect the completeness and representativeness of the collected data.
3. **Model Assumptions and Simplifications:** The calibration, forecasting, and fusion models rely on specific assumptions and simplifications about the underlying data patterns and relationships. Deviations from these assumptions, such as abrupt changes in environmental conditions or sensor behavior, may impact the models' accuracy and predictive performance.

4. **Data Acquisition Frequency:** The frequency of sensor data acquisition may vary, influencing the temporal resolution of the forecasts and the ability to capture rapid changes in CO_2 levels. Lower data acquisition frequencies could limit the models' ability to respond promptly to short-term fluctuations.
5. **External Factors and Interference:** External factors, such as changes in weather patterns or human activities, may introduce interference or noise into the sensor data. These external influences could potentially obscure the true underlying trends and patterns in CO_2 levels.

These delimitations acknowledge the inherent constraints of the study and provide context for interpreting the findings within these limitations.

1.7 Structure of the thesis

Chapter 2 reviews existing methods in outlier detection, sensor calibration, short-term prediction, and sensor fusion, laying the groundwork for the techniques developed in this thesis. Chapter 3 provides a detailed explanation of the methodologies used, including the construction and application of the Deep Calibration Method (DeepCM), Long Short-Term Memory (LSTM) models, and Gaussian Process Regression (GPR) models for sensor fusion. Chapter 4 discusses the implementation of these methods, covering the practical steps for sensor calibration, CO2 level prediction, and multi-sensor data fusion, and offers a technical overview of the conducted experiments. Chapter 5 presents and analyzes the results of the implemented methods, comparing the performance of different models for sensor calibration, short-term forecasting, and sensor fusion, and quantifying the improvements in prediction accuracy and reliability. Chapter 6 concludes the thesis by summarizing the findings, evaluating the limitations of the study, and offering recommendations for future research, particularly in enhancing sensor accuracy, long-term prediction, and data integration techniques.

Chapter 2

Background

There are three main work elements in the research: sensor calibration, sensor data short-term prediction, and sensor fusion. However, before starting the research in these three areas, I also explored methods for outlier detection and processing. Therefore, in this section I will present existing research work in these three areas as well as in sensor calibration methods.

2.1 Outlier Detection

2.1.1 Statistical Methods for Outlier Detection

Researchers have proposed various statistical methods to detect outliers, which are observations that deviate significantly from the rest of the data distribution. These methods are essential in data pre-processing and quality assurance across various domains, including finance, healthcare, and manufacturing.

1. Z-score Method

The Z-score method is a widely used statistical technique that measures how many standard deviations a data point x_i is from the mean μ of the dataset $\{x_1, x_2, \ldots, x_n\}$.[11] It is defined as:

$$Z_i = \frac{x_i - \mu}{\sigma} \tag{2.1}$$

where μ is the sample mean:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.2}$$

and σ is the sample standard deviation:

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu)^2} \tag{2.3}$$

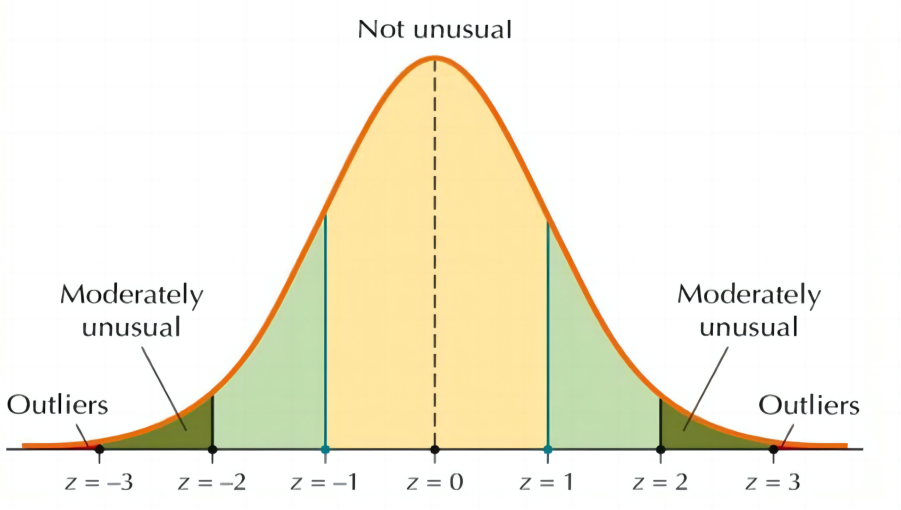


Figure 2.1.1: Z-core outlier detection

The Z-score indicates how far a data point deviates from the mean in terms of standard deviations. Typically, data points with $|Z_i| > 2$ or $|Z_i| > 3$ are considered outliers, depending on the chosen threshold. An example of Z-core outlier detection is shown in figure 2.1.1.

2. Boxplot Analysis

Boxplots provide a graphical summary of the distribution of a dataset. They display key statistical measures such as the median (Q2), the first quartile (Q1), the third quartile (Q3), and the interquartile range (IQR). The IQR is calculated as:

$$\text{IQR} = Q3 - Q1 \tag{2.4}$$

In boxplot analysis, outliers are identified as points that fall outside the range:

$$[Q1 - 1.5 \times \text{IQR}, Q3 + 1.5 \times \text{IQR}] \tag{2.5}$$

Points beyond these whiskers are marked as potential outliers, which may warrant further investigation.

3. Grubbs' Test

Grubbs' test is a statistical test used to detect a single outlier in a univariate dataset.[12] It is based on the hypothesis that one data point x_i is an outlier. The test statistic G for x_i is calculated as:

$$G = \frac{|x_i - \mu|}{s} \tag{2.6}$$

where μ is the sample mean and s is the sample standard deviation.

The critical value $G_{\alpha,n-1}$, derived from the Student's t-distribution with $n - 1$ degrees of freedom, is compared against G to determine if x_i is an outlier. For a significance level α, if $G > G_{\alpha,n-1}$, x_i is considered an outlier.

2.1.2 Distance and Density-based Methods

Another common approach to outlier detection involves using distance or density metrics. These methods are particularly useful in scenarios where the data distribution is not Gaussian or where traditional statistical methods may not be effective.

1. k-Nearest Neighbors (KNN) Method

The k-nearest neighbors (KNN) method is a distance-based approach for detecting outliers by assessing the proximity of data points to their k nearest neighbors.[13]

For a data point x_i, let $N_k(x_i)$ denote its k-nearest neighbors based on a distance metric $d(x_i, x_j)$. The outlier score O_i for x_i is calculated as:

$$O_i = \frac{1}{k} \sum_{j \in N_k(x_i)} d(x_i, x_j) \tag{2.7}$$

The score O_i is the average distance between x_i and its k nearest neighbors. Intuitively, outliers have larger distances to their neighbors compared to non-outliers.

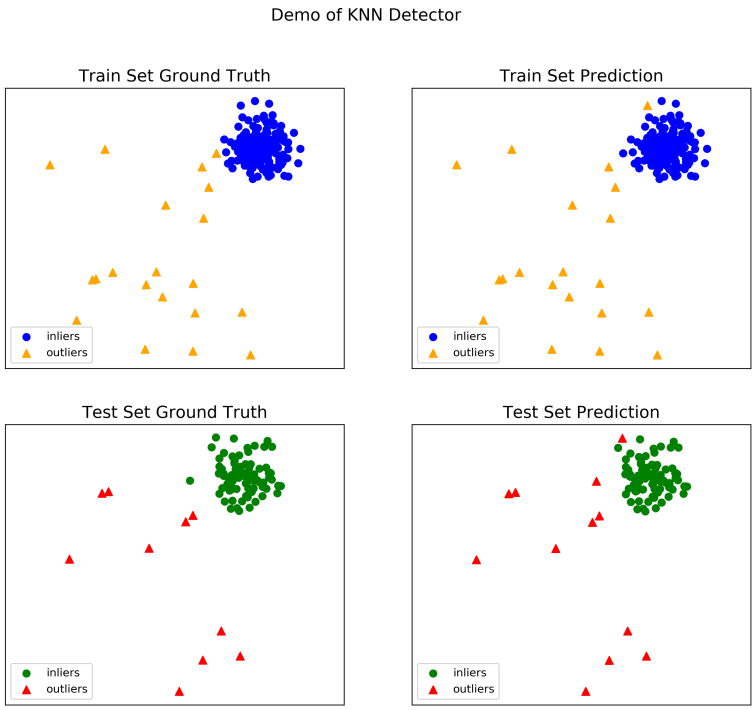


Figure 2.1.2: knn outlier detection

To identify outliers using KNN:

- Calculate O_i for each data point x_i.
- Compare O_i to the average score of its neighbors.

- Data points with O_i significantly higher than the average are potential outliers.

A demo ofKNN outlier detction is shown in figure 2.1.2. KNN is effective in detecting outliers in datasets where outliers are isolated and distant from the majority of data points. It is robust to varying data distributions and can handle high-dimensional data.

Consider a dataset where KNN is applied to detect outliers. By computing O_i for each data point and setting a threshold based on the distribution of scores, outliers that deviate significantly in proximity from their k nearest neighbors can be identified and analyzed further.

2. Local Outlier Factor (LOF)

The Local Outlier Factor (LOF) is a method for detecting outliers that measures the local density deviation of a data point with respect to its neighbors. It evaluates how isolated or different a data point x_i is compared to its local neighborhood.[14]

For a data point x_i, let $N_k(x_i)$ denote its k-nearest neighbors. The LOF score $\text{LOF}(x_i)$ is calculated as:

$$\text{LOF}(x_i) = \frac{\sum_{j \in N_k(x_i)} \frac{\text{density}(x_j)}{\text{density}(x_i)}}{k} \tag{2.8}$$

where $\text{density}(x_i)$ denotes the local density of x_i, estimated based on the distances to its k-nearest neighbors.

The numerator sums the ratio of densities between x_i and its neighbors x_j, indicating how much denser the neighbors are compared to x_i. Dividing by k normalizes this value. A high LOF score $\text{LOF}(x_i) > 1$ suggests that x_i has a lower density than its neighbors, indicating that x_i is potentially an outlier. The higher the LOF score, the more likely x_i is considered an outlier relative to its local neighborhood. LOF is effective in identifying outliers in datasets with varying densities and complex geometric structures. It can handle datasets where outliers may not be well-separated in the global space but stand out in their local regions.

Consider a dataset where LOF is applied to detect outliers. By computing LOF scores for each data point and comparing them against a threshold, outliers that deviate significantly from their local neighborhoods can be identified and investigated further. An example of LOF outlier detection is shown in figure 2.1.3.

3. Isolation Forest

Isolation Forest is an ensemble learning method designed to isolate outliers in a dataset by leveraging the principle that anomalies are typically few and different. It constructs isolation trees through random partitioning of data points until each point is isolated in its own leaf node.[15]

The anomaly score $s(x_i)$ for a data point x_i in Isolation Forest is computed based on the path length $h(x_i)$ in the tree:

$$s(x_i) = 2^{-\frac{E(h(x_i))}{c(n)}} \tag{2.9}$$

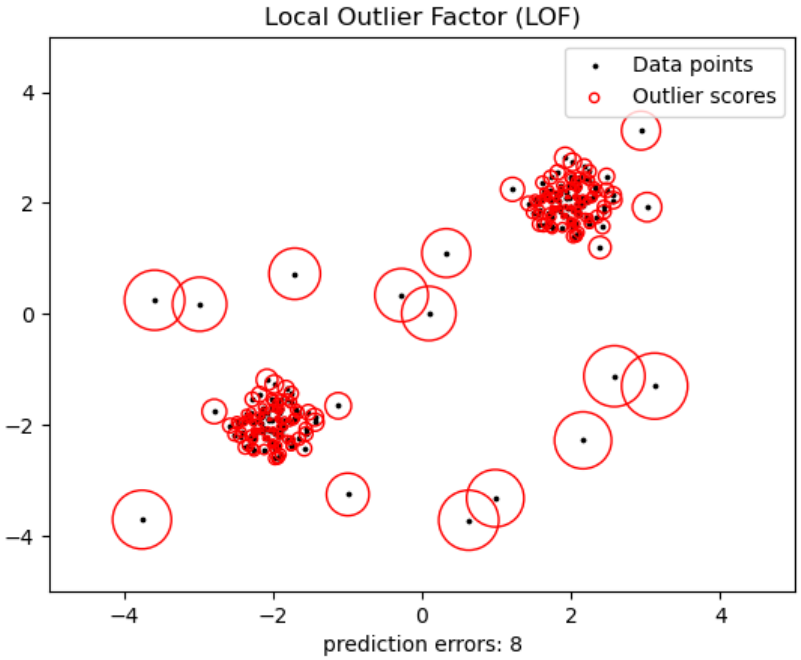


Figure 2.1.3: LOF outlier detection

where:

- $E(h(x_i))$ is the average path length for x_i across all isolation trees.
- $c(n)$ is a normalization factor, typically $c(n) = 2H(n-1)/(n-1)$, where $H(n-1)$ is the harmonic number.
- $h(x_i)$ is the path length in a single isolation tree, representing the number of edges traversed from the root to isolate x_i.

Lower anomaly scores $s(x_i)$ indicate a higher likelihood of x_i being an outlier. Isolation Forest identifies outliers as data points with shorter average path lengths across multiple trees, suggesting they are easier to isolate due to their distinctiveness. Isolation Forest is effective in high-dimensional datasets and can efficiently handle large datasets. It does not require assumptions about the underlying data distribution and can detect outliers quickly by leveraging random partitioning.

Consider applying Isolation Forest to a dataset with high-dimensional features. By computing $s(x_i)$ for each data point and setting a threshold based on the distribution of scores, outliers that have shorter path lengths across the trees can be identified as potential anomalies.

2.1.3 Machine Learning Approaches in Outlier Detection

Machine learning approaches, such as Support Vector Machines (SVM) and clustering methods, are widely used for outlier detection tasks in various domains.

1. Support Vector Machines (SVM)

Support Vector Machines (SVM) are powerful tools for outlier detection by maximizing the margin between normal and abnormal points through the construction of an optimal hyperplane.

For a binary classification problem, SVM aims to find a hyperplane defined by $\mathbf{w}\cdot\mathbf{x}+b = 0$ that maximizes the margin $\frac{1}{\|\mathbf{w}\|}$ between the two classes. Outliers are identified as data points that either lie outside this margin or are misclassified.[16]

The primal formulation of the SVM problem is given by:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \tag{2.10}$$

subject to:

$$y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad \text{for } i = 1,\ldots,n \tag{2.11}$$

where:

- $\mathbf{w}$ is the weight vector perpendicular to the hyperplane,
- b is the bias term,
- ξ_i are slack variables that allow for soft-margin classification,
- y_i are class labels ($y_i = \pm 1$), indicating normal or abnormal classes,
- C is a regularization parameter that controls the trade-off between maximizing the margin and minimizing the classification error.

To detect outliers using SVM:

- Train an SVM model on labeled data where outliers are appropriately labeled or identified.
- During training, SVM learns the optimal hyperplane that separates the majority of normal points from potential outliers.
- After training, compute the distances or margins from the hyperplane for each data point.
- Data points that lie significantly beyond the margin or violate the margin constraint are classified as outliers.

SVM offers several advantages in outlier detection:

- It can handle high-dimensional data effectively.
- SVM is robust to overfitting due to the regularization parameter C.
- It can capture complex relationships in data, making it suitable for non-linear separation using kernel methods.

2. Clustering Methods

Clustering methods are unsupervised techniques used to group data points into clusters based on their similarity. Outliers are data points that deviate significantly from the clustered patterns and do not belong well to any identified cluster.[17]

Clustering algorithms such as k-means, DBSCAN (Density-Based Spatial Clustering of Applications with Noise), and hierarchical clustering aim to partition the dataset into groups or clusters.[18] Outliers are typically defined as points that are distant from the centroids of clusters or have low density in density-based methods.

The k-means algorithm aims to partition n data points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ into k clusters $C_1, C_2, \ldots, C_k$ such that the within-cluster variance is minimized. The objective function is:

$$\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \mu_i\|^2 \tag{2.12}$$

where μ_i is the centroid of cluster C_i, and $\mathbf{S} = \{C_1, C_2, \ldots, C_k\}$ represents the partition.

Outliers in k-means can be identified as points that do not fit well into any cluster or are far from the centroids.

DBSCAN is a density-based clustering method that identifies clusters as regions of high density separated by regions of low density. It defines clusters as maximal sets of density-connected points and identifies outliers as points that do not belong to any cluster.

Hierarchical clustering builds a hierarchy of clusters either bottom-up (agglomerative) or top-down (divisive). Outliers can be detected in hierarchical clustering as data points that do not fit well into any level of the hierarchy or are far from the centers of clusters at each level.

To detect outliers using clustering methods:

- Apply a clustering algorithm to partition the data into clusters.
- Evaluate each data point based on its distance from the cluster centroids or its density within its neighborhood.
- Points that are distant from cluster centers or have low density can be classified as outliers.

An example of clustering method outlier detection is shown in figure 2.1.4. The image depicts a two-dimensional space with a cartesian plane (X and Y axis) as a reference system. The plane contains three clusters of black points, each enclosed within a green circle, indicating their grouping. These clusters likely represent data points with similar characteristics.

A single red point, labeled ”Outlier,” is located at a significant distance from all three clusters. Dotted lines connect this outlier to each of the clusters. This visualization illustrates the concept of an outlier in data analysis—a data point that

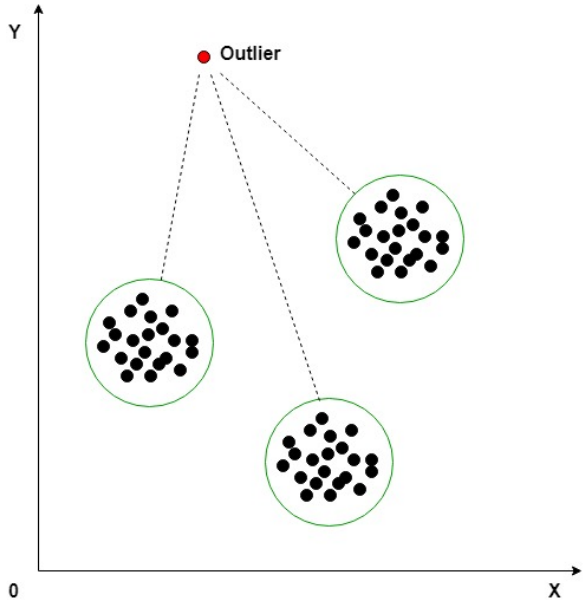


Figure 2.1.4: Clustering method outlier detection

deviates significantly from the other data points and does not belong to any particular group.

Clustering methods offer several advantages for outlier detection:

- They are effective in identifying outliers in high-dimensional datasets.
- Clustering can capture complex relationships and structures in data.
- These methods do not require labeled data and can handle large datasets efficiently.

2.1.4 Outlier Detection in Streaming Data

Detecting outliers in streaming data poses unique challenges due to the need for real-time and continuous processing. Techniques designed for streaming data typically employ window-based methods and drift detection approaches to identify anomalies effectively.

1. Window-based Methods

Window-based methods process data in consecutive windows or segments, where each window represents a subset of the streaming data. Outliers are detected by analyzing the statistical properties within each window.[19]

Let $W_t = \{x_{t-w+1}, x_{t-w+2}, \ldots, x_t\}$ denote a sliding window of size w at time t, containing the most recent w data points in the stream. Outliers are detected by computing statistics such as mean, standard deviation, or other distribution parameters within W_t.

One common approach is the Moving Z-score method, which calculates the Z-score for

each data point x_t within the window W_t:

$$Z_t = \frac{x_t - \mu_{W_t}}{\sigma_{W_t}} \tag{2.13}$$

where μ_{W_t} and σ_{W_t} are the mean and standard deviation of W_t, respectively. Data points with $|Z_t|$ exceeding a threshold θ are considered outliers.

2. Drift Detection Methods

Drift detection methods monitor changes in the data distribution over time to detect anomalies that deviate from the expected behavior.[20]

Drift detection algorithms compare the current data distribution with a reference distribution or historical data to identify significant deviations. Methods such as Cumulative Sum (CUSUM) and Change Detection have been adapted for streaming data to detect abrupt changes indicative of outliers.[21]

CUSUM detects changes in the mean of the streaming data by calculating cumulative sums of deviations from the mean over time. The CUSUM statistic S_t at time t is updated recursively as:

$$S_t = \max(0, S_{t-1} + (x_t - \mu) - k) \tag{2.14}$$

where μ is the reference mean, k is a threshold parameter, and x_t is the current data point. Anomaly detection occurs when S_t exceeds a predefined threshold.

To detect outliers in streaming data:

- Implement window-based methods to analyze data within sliding windows and identify outliers based on statistical deviations.
- Utilize drift detection algorithms to monitor changes in data distribution and detect anomalies indicative of outliers.
- Continuously update models and thresholds to adapt to evolving data patterns and ensure real-time detection of anomalies.

Outlier detection in streaming data offers several advantages:

- Real-time detection enables prompt response to anomalies and potential threats.
- Continuous monitoring ensures that outliers are identified as soon as they occur, minimizing potential risks.
- Adaptive algorithms can adjust to changing data dynamics and maintain effectiveness over time.

2.1.5 Advancements with Deep Learning

Recent advancements in outlier detection have seen the application of deep learning techniques, particularly Autoencoders and Generative Adversarial Networks (GANs), which leverage their ability to model complex data distributions.

1. Autoencoders

Autoencoders are neural networks designed to learn efficient representations of input data by encoding them into a lower-dimensional space and then decoding them back to the original dimension. They can be used effectively for outlier detection by reconstructing data points and identifying those that deviate significantly from the learned data distribution.[22]

Consider an autoencoder with an encoder function $f_\theta : \mathbb{R}^d \to \mathbb{R}^h$ and a decoder function $g_{\theta'} : \mathbb{R}^h \to \mathbb{R}^d$, where θ and θ' are the parameters of the encoder and decoder, respectively. The reconstruction error for a data point $\mathbf{x} \in \mathbb{R}^d$ is given by:

$$\mathcal{L}(\mathbf{x}) = \|\mathbf{x} - g_{\theta'}(f_\theta(\mathbf{x}))\|^2 \tag{2.15}$$

Autoencoders detect outliers by computing the reconstruction error for each data point. High reconstruction errors indicate that the data point does not conform well to the learned patterns, suggesting it may be an outlier.

2. Generative Adversarial Networks (GANs)

GANs consist of two neural networks: a generator G and a discriminator D. The generator learns to generate synthetic data that mimics the distribution of normal data points, while the discriminator distinguishes between real and generated data. GANs can be adapted for outlier detection by training the discriminator to detect deviations from the normal data distribution.[23]

Let $\mathbf{z} \sim p(\mathbf{z})$ be a latent vector sampled from a prior distribution $p(\mathbf{z})$. The generator G maps $\mathbf{z}$ to a synthetic data point $\tilde{\mathbf{x}} = G(\mathbf{z})$. The discriminator D outputs a probability $D(\mathbf{x})$ indicating whether a data point $\mathbf{x}$ is real (from the normal distribution) or fake (generated by G).

The training objective for GANs is:

$$\min_G \max_D \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))] \tag{2.16}$$

where $p_{\text{data}}(\mathbf{x})$ is the true data distribution.

Outliers are detected by observing the discriminator's output: data points with low $D(\mathbf{x})$ values are likely to be outliers, as they are less similar to the normal data distribution learned by the discriminator.

To detect outliers using deep learning techniques:

- Train an autoencoder or GAN on normal data to learn the underlying data distribution.
- Compute reconstruction errors for autoencoders or discriminator outputs for GANs to identify outliers.
- Adjust model parameters and thresholds based on validation data to optimize outlier detection performance.

Deep learning techniques offer several advantages in outlier detection:

- They can capture intricate data patterns and distributions, enhancing the detection of subtle anomalies.
- Autoencoders and GANs can adapt to non-linear and complex data structures, improving outlier detection accuracy.
- These methods can be trained in an unsupervised manner, requiring minimal labeled data.

2.2 Sensor Calibration

Sensor calibration is crucial to ensure the accuracy and reliability of sensor outputs. Calibration aims to determine the measurement errors of sensors and reduce these errors through correction formulas, tables, or calibration curves. This part introduces six commonly used sensor calibration methods and their theoretical basis.

2.2.1 Single-point Calibration

Single-point calibration is a straightforward calibration technique that involves calibrating the sensor at a single known standard point. This method operates under the assumption that the sensor response is linear near the calibration point. The primary advantage of single-point calibration is its simplicity and ease of implementation, making it a popular choice for initial sensor calibration in various applications.[24]

The theoretical basis of single-point calibration can be described as follows. Suppose we have a sensor whose output signal is denoted by V_{out}. When the sensor is exposed to a known input signal X_{ref}, the corresponding output V_{ref} is recorded. The relationship between the input and the output can be expressed as a linear function:

$$V_{\text{out}} = aX + b \tag{2.17}$$

where a is the sensitivity or gain of the sensor, and b is the offset or bias. The goal of the single-point calibration is to determine these parameters a and b based on the measurement at the reference point $(X_{\text{ref}}, V_{\text{ref}})$.[25]

Assuming the sensor response is approximately linear near the calibration point, the sensitivity a can be approximated by:

$$a \approx \frac{V_{\text{ref}} - V_{\text{o}}}{X_{\text{ref}} - X_0} \tag{2.18}$$

where V_{o} is the output voltage corresponding to a baseline input signal X_0. In many cases, X_0 can be zero, simplifying the equation to:

$$a \approx \frac{V_{\text{ref}}}{X_{\text{ref}}} \tag{2.19}$$

Once the sensitivity a is determined, the offset b can be calculated using the known reference point:

$$b = V_{\text{ref}} - aX_{\text{ref}} \tag{2.20}$$

Thus, the calibrated output V_{cal} for any input signal X can be obtained by:

$$V_{\text{cal}} = aX + b \tag{2.21}$$

In summary, single-point calibration provides a means to estimate the sensor parameters based on a single known standard point. This method is particularly effective for sensors with a linear or approximately linear response near the calibration point. However, it may not be suitable for sensors with significant non-linearity or for applications requiring high accuracy over a wide range of input signals. In such cases, multi-point calibration or other advanced calibration techniques may be more appropriate.

Despite its limitations, single-point calibration remains a valuable tool for many practical applications due to its simplicity and ease of implementation. By ensuring the sensor is operating correctly at a known point, it provides a quick and efficient way to achieve acceptable accuracy in many scenarios.

2.2.2 Multi-point Calibration

Multi-point calibration involves measurements at multiple known standard points and fitting a calibration curve using methods such as least squares fitting. The theoretical basis is to establish an input-output relationship model by using a series of known input signals and corresponding sensor outputs.[26] Multi-point calibration significantly improves the measurement accuracy of nonlinear sensors by generating calibration curves through polynomial fitting or interpolation.

The goal of multi-point calibration is to minimize the error between the actual sensor output and the predicted output based on a model. Given a set of calibration points (x_i, y_i), where x_i are the known input values and y_i are the corresponding sensor outputs, we aim to find a calibration function $f(x)$ that best fits these points. A common choice for $f(x)$ is a polynomial of degree n:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{2.22}$$

The coefficients $a_0, a_1, \ldots, a_n$ are determined by minimizing the sum of the squared differences between the observed outputs and the outputs predicted by the polynomial. This is formulated as:

$$\min_{a_0,a_1,\ldots,a_n} \sum_{i=1}^{m} (y_i - f(x_i))^2 \tag{2.23}$$

The least squares method provides a way to determine the coefficients of the polynomial. For a polynomial of degree n, we need to solve the following system of linear equations derived from the condition of minimization:

$$\mathbf{X}^\top \mathbf{X}\mathbf{a} = \mathbf{X}^\top \mathbf{y} \tag{2.24}$$

where $\mathbf{X}$ is the Vandermonde matrix of the input values:

$$\mathbf{X} = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{pmatrix} \tag{2.25}$$

$\mathbf{y}$ is the vector of observed outputs:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \tag{2.26}$$

and $\mathbf{a}$ is the vector of coefficients:

$$\mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} \tag{2.27}$$

The solution to the system $\mathbf{X}^\top \mathbf{X}\mathbf{a} = \mathbf{X}^\top \mathbf{y}$ gives the coefficients $\mathbf{a}$.

Polynomial fitting involves finding a polynomial that passes as close as possible to the set of calibration points. For a polynomial of degree n, the least squares fitting approach provides a robust method to determine the coefficients that minimize the overall error. In some cases, especially when dealing with highly nonlinear sensor responses, polynomial interpolation can be used. Interpolation ensures that the polynomial passes exactly through all the calibration points.

The interpolation polynomial can be expressed using the Lagrange form:

$$P(x) = \sum_{i=1}^{m} y_i \prod_{\substack{1 \le j \le m \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \tag{2.28}$$

In practice, the multi-point calibration process involves the following steps:

1. **Selection of Calibration Points**: Choose multiple known standard points that cover the range of input values for the sensor.
2. **Measurement**: Record the sensor outputs corresponding to the selected input values.
3. **Fitting**: Use least squares fitting to determine the coefficients of the polynomial that best fits the data.
4. **Validation**: Validate the calibration curve by comparing the predicted outputs with actual sensor outputs for additional test points.

By following these steps, multi-point calibration can significantly enhance the accuracy of sensors, especially those with nonlinear characteristics. This approach is widely used in various fields, including metrology, instrumentation, and control systems, to ensure precise and reliable measurements.

2.2.3 Comparison Calibration

Comparison calibration involves using a certified high-precision standard device as a reference to correct and adjust the measurements of a sensor under test. This method is crucial in fields such as temperature and pressure measurements to ensure accurate and reliable readings.

The fundamental principle of comparison calibration revolves around the concept of minimizing the error between the sensor's output x_i and the standard device's output y_i. The sensor's output is typically corrected using a calibration equation derived from the comparison measurements. Suppose we have n pairs of measurements (x_i, y_i), where x_i represents the sensor's measurement and y_i represents the standard device's measurement.

A common approach is to use a linear calibration model[27]:

$$x_{\text{calibrated}} = a \cdot y + b \tag{2.29}$$

where:

- $x_{\text{calibrated}}$ is the calibrated measurement from the sensor,
- y is the measurement from the standard device,
- a and b are calibration coefficients.

To determine a and b, typically, least squares regression is applied to minimize the squared differences between x_i and $a \cdot y_i + b$:

$$\min_{a,b} \sum_{i=1}^{n} (x_i - a \cdot y_i - b)^2 \tag{2.30}$$

The coefficients a and b can be computed as follows:

$$a = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} y_i^2 - (\sum_{i=1}^{n} y_i)^2} \tag{2.31}$$

$$b = \frac{\sum_{i=1}^{n} x_i - a\sum_{i=1}^{n} y_i}{n} \tag{2.32}$$

For instance, in temperature calibration, if x_i represents the readings from a thermocouple sensor and y_i represents the readings from a platinum resistance thermometer (PRT), the calibration equation could be[28]:

$$x_{\text{calibrated}} = a \cdot y + b \tag{2.33}$$

where a and b are determined through the comparison of measurements between the thermocouple and the PRT across different temperature ranges.

2.2.4 Environmental Calibration

Environmental calibration is essential for improving sensor accuracy by considering the influence of environmental conditions such as temperature T, humidity H, and pressure P on sensor output. This section explores the theoretical foundations and mathematical formulations used in environmental calibration.[29]

The sensor's response y to an input x can be affected by environmental variables T, H, and P. We can express this relationship as:

$$y = f(x, T, H, P) + e \tag{2.34}$$

where:

- f is the function describing the sensor's response to x and environmental conditions.
- e represents the error term incorporating random noise and systematic deviations due to environmental factors.

To mitigate the influence of environmental conditions on sensor measurements, the next two steps need to be done:

1. **Sensor Response Model Incorporating Environmental Effects:**

   The sensor's output y can be modeled as:

   $$y = f(x, T, H, P) + e \tag{2.35}$$

   Here, f encapsulates how x and environmental variables T, H, P influence y.

2. **Error Compensation Model:**

   To compensate for environmental effects and improve measurement accuracy, an error model $\hat{e}(T, H, P)$ is derived from calibration experiments:

   $$y_{\text{calibrated}} = y - \hat{e}(T, H, P) \tag{2.36}$$

This model adjusts y based on estimated environmental errors $\hat{e}$.

Understanding the impact of each environmental variable T, H, P involves:

- **Sensitivity Analysis:** Quantifying how changes in T, H, P affect y.
- **Modeling Environmental Errors:** Developing mathematical relationships or empirical models to predict and compensate for these effects.

Environmental calibration utilizes theoretical models and empirical data to mitigate the influence of environmental conditions on sensor measurements. By understanding and quantifying these effects, engineers can enhance the reliability and accuracy of sensors in various operational environments.

2.2.5 Dynamic Calibration

Dynamic calibration evaluates and adjusts the sensor's response characteristics to rapidly changing signals. The theoretical basis is to analyze and correct the sensor's time-domain and frequency-domain response by applying known dynamic signals with specific frequencies and amplitudes.[29] Dynamic calibration is commonly used for vibration sensors, accelerometers, and applications requiring accurate measurement of dynamic changes.

Dynamic calibration involves analyzing the frequency response of sensors to ensure accurate measurement of dynamic signals. Here, we'll explore the concept of transfer functions and their role in dynamic calibration.

Transfer Function Representation

The transfer function $H(s)$ relates the output $Y(s)$ to the input $X(s)$ in the Laplace domain:

$$Y(s) = H(s) \cdot X(s) \tag{2.37}$$

where $s = \sigma + j\omega$ is the complex frequency variable (σ represents the real part and ω represents the angular frequency).

Frequency Response Characteristics

The frequency response $H(j\omega)$ of the sensor describes how the sensor responds to different frequencies of the input signal $x(t)$:

$$Y(j\omega) = H(j\omega) \cdot X(j\omega) \tag{2.38}$$

where $X(j\omega)$ and $Y(j\omega)$ are the Fourier transforms of $x(t)$ and $y(t)$, respectively.

In dynamic calibration, dynamic signals $x(t)$ are often modeled using sinusoidal signals with varying frequencies and amplitudes. For instance, a sinusoidal input $x(t) = A\sin(\omega t + \phi)$ can be used to characterize the sensor's response across different frequencies.

To estimate the transfer function $H(s)$, system identification techniques such as Fourier analysis, spectral analysis, or frequency response analysis are employed. These techniques involve:

- **Frequency Sweep:** Applying sinusoidal signals with varying frequencies to the sensor and measuring the resulting output.
- **Fourier Transform:** Computing the Fourier transform of the input and output signals to determine the frequency response $H(j\omega)$.
- **System Modeling:** Using identified frequency response data to fit a transfer function model $H(s)$ that best represents the sensor's behavior.

Let's consider a simplified example where the sensor's transfer function is modeled as a second-order system:

$$H(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{2.39}$$

where ω_n is the natural frequency of the sensor's response and ζ is the damping ratio.

Once the transfer function $H(s)$ is estimated, adjustments to the sensor's calibration parameters can be made to optimize its performance. This involves:

- **Gain Adjustment:** Scaling the amplitude response to match the expected output for different input levels.
- **Phase Compensation:** Correcting phase shifts introduced by the sensor's response to ensure accurate time-domain representation.
- **Frequency Bandwidth Adjustment:** Optimizing the sensor's frequency response characteristics, such as resonance frequencies and bandwidth, for specific applications.

Dynamic calibration through transfer function analysis provides a systematic approach to ensuring sensors accurately capture and respond to dynamic signals. By understanding and adjusting the sensor's frequency-domain characteristics, engineers can enhance the sensor's reliability and precision in measuring dynamic changes across various applications.

This mathematical framework underscores the importance of dynamic calibration in sensor technology, bridging theoretical insights with practical applications for optimal performance.

2.2.6 Machine Learning and Deep Learning-based Calibration

Machine learning (ML) and deep learning (DL) techniques are increasingly applied to sensor calibration due to their capability to model complex nonlinear relationships and intricate environmental dependencies.Common machine learning methods include regression analysis, support vector machines (SVM), and decision trees; common deep learning methods include deep neural networks (DNN), convolutional neural networks (CNN), and recurrent neural networks (RNN). These methods enable the construction

of models that accurately predict sensor outputs based on extensive training data collected under various environmental conditions.

1. Regression Analysis

Regression methods, such as linear regression, provide a straightforward approach to modeling the relationship between sensor inputs and outputs. In a simple linear regression model:

$$y = \beta_0 + \beta_1 x + \epsilon \tag{2.40}$$

where y represents the sensor output, x is the sensor input, β_0 and β_1 are coefficients, and ϵ denotes the error term. Linear regression assumes a linear relationship between variables, which may be suitable for certain sensor types with relatively simple input-output mappings.

2. Support Vector Machines (SVM)

Support Vector Machines (SVM) extend regression capabilities to more complex relationships by seeking a hyperplane that maximizes the margin between data points of different classes.[30] For regression tasks, SVM constructs a hyperplane to predict continuous values:

$$y = \sum_{i=1}^{n} \alpha_i K(x_i, x) + b \tag{2.41}$$

where α_i are coefficients determined during training, K is the kernel function that computes similarity between inputs x_i and x, and b is a bias term. SVMs are effective in capturing nonlinear relationships and are robust against overfitting when appropriately tuned.

3. Decision Trees

Decision trees partition data based on input features, making them particularly useful for sensor calibration across various environmental conditions.[31] A decision tree predicts y by recursively partitioning the input space:

$$y = \sum_{j=1}^{m} c_j I(x \in R_j) \tag{2.42}$$

where c_j are constants associated with each leaf node R_j, and I is an indicator function that evaluates whether x belongs to region R_j. Decision trees are intuitive and capable of handling complex interactions between sensor inputs and outputs.

These methods form the foundational tools in traditional machine learning, providing robust frameworks for capturing and modeling sensor behavior across varying environmental conditions.

4. Deep Neural Networks (DNN)

Moving beyond traditional methods, deep neural networks (DNNs) offer enhanced capabilities in capturing intricate sensor dependencies. DNNs consist of multiple layers of interconnected neurons and compute outputs through successive layers.[32] Each layer transforms the input data using weights and biases, followed by activation functions to introduce non-linearities:

$$y = f(\mathbf{W}^{(L)} f(\mathbf{W}^{(L-1)} \dots f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \dots) + \mathbf{b}^{(L)} \tag{2.43}$$

where $\mathbf{x}$ is the input vector, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are weights and biases of layer l, and f denotes the activation function applied element-wise. DNNs excel in learning complex mappings between sensor inputs and outputs but require sufficient data and computational resources for training.

5. Convolutional Neural Networks (CNN)

For spatial data, convolutional neural networks (CNNs) excel in extracting meaningful features through convolutional layers.[33]CNNs compute outputs by convolving inputs with learnable filters and applying activation functions to the resulting feature maps:

$$y = f(\mathbf{W}^{(L)} * f(\mathbf{W}^{(L-1)} * \dots f(\mathbf{W}^{(1)} * \mathbf{x} + \mathbf{b}^{(1)}) \dots) + \mathbf{b}^{(L)} \tag{2.44}$$

where $*$ denotes the convolution operation. CNNs are particularly effective for sensor data that exhibits spatial correlations, such as image or spatial sensor inputs.

6. Recurrent Neural Networks (RNN)

Handling sequential data, recurrent neural networks (RNNs) process inputs iteratively through recurrent connections.[34] This iterative approach allows RNNs to capture temporal dependencies, making them suitable for time-series sensor data:

$$y_t = f(\mathbf{W}^{(L)} y_{t-1} + \mathbf{W}^{(1)}\mathbf{x}_t + \mathbf{b}^{(1)} + \mathbf{b}^{(L)}) \tag{2.45}$$

where y_t is the output at time t, $\mathbf{x}_t$ is the input at time t, and $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are weights and biases. RNNs are adept at capturing dynamic patterns and long-term dependencies in sequential sensor data.

In summary, machine learning and deep learning methods provide powerful tools for sensor calibration, allowing for the modeling of complex sensor behaviors and environmental interactions. Regression analysis, SVMs, decision trees, DNNs, CNNs, and RNNs each offer unique strengths in capturing and predicting sensor outputs based on input data. The choice of method depends on the nature of the sensor data, the complexity of relationships, and the computational resources available for model training and deployment.

2.3 Sensor Short-term Forecast

2.3.1 Statistical Methods

1. Simple Exponential Smoothing (SES)

SES is suitable for data without seasonality or trends. The forecast $\hat{Y}_{t+1}$ is computed as:

$$\hat{Y}_{t+1} = \alpha Y_t + (1 - \alpha)\hat{Y}_t \tag{2.46}$$

where Y_t is the actual observation at time t, $\hat{Y}_t$ is the previous forecast, and α is the smoothing parameter (typically between 0 and 1).[35]

2. Double Exponential Smoothing (DES)

DES considers the trend in time series data. The forecast $\hat{Y}_{t+1}$ includes both trend and level components:

$$\hat{Y}_{t+1} = \hat{Y}_t + b_t + \alpha(Y_t - \hat{Y}_t) \tag{2.47}$$

$$b_t = \beta(\hat{Y}_t - \hat{Y}_{t-1}) + (1 - \beta)b_{t-1} \tag{2.48}$$

where b_t represents the trend component and β is the trend smoothing parameter.[36]

Exponential Smoothing methods, such as SES and DES, offer valuable tools for time series forecasting by leveraging past observations to predict future trends. SES focuses on updating forecasts based on recent data points and a smoothing parameter α that controls the influence of each observation. In contrast, DES extends SES by incorporating a trend component, b_t, which captures directional changes in the data over time. Both methods emphasize adaptability, adjusting forecasts dynamically as new data becomes available.

1. **Exponential Smoothing Principle**: Both SES and DES operate on the principle of exponentially decreasing weights applied to observations as they recede into the past, with α and β determining the rate of decay for level and trend components, respectively.
2. **Initial Conditions**: The initial values $\hat{Y}_1$ and b_1 must be specified or estimated before applying the models. These values significantly influence subsequent forecasts.
3. **Forecast Accuracy**: The accuracy of forecasts depends on the appropriateness of the smoothing parameters α and β, which should be chosen through model validation techniques such as cross-validation or optimization procedures based on forecasting accuracy metrics (e.g., Mean Squared Error).
4. **Model Adaptation**: SES and DES are adaptive models in the sense that they update forecasts as new data becomes available, making them suitable for dynamic forecasting environments.

2.3.2 Machine Learning Methods

Machine learning approaches include various algorithms such as Autoregressive Integrated Moving Average (ARIMA), Seasonal ARIMA (SARIMA), etc., which are designed to capture autocorrelation and seasonality in time series data.

1. ARIMA Model

The Autoregressive Integrated Moving Average (ARIMA) model can be represented as:

(a) Autoregressive (AR) Component

$$X_t = c + \phi_1 X_{t-1} + \phi_2 X_{t-2} + \ldots + \phi_p X_{t-p} + \epsilon_t \tag{2.49}$$

where X_t is the value of the time series at time t, c is a constant, $\phi_1, \phi_2, \ldots, \phi_p$ are the autoregressive coefficients, and $\epsilon_t \sim N(0, \sigma^2)$ is white noise.

(b) Integrated (I) Component

$$Y_t = (1 - B)^d X_t \tag{2.50}$$

where B is the backshift operator (i.e., $B^d X_t = X_{t-d}$), and d is the order of differencing required to achieve stationarity.

(c) Moving Average (MA) Component

$$X_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} \tag{2.51}$$

where μ is the mean of the time series, ϵ_t are white noise error terms, and $\theta_1, \theta_2, \ldots, \theta_q$ are the moving average coefficients.[37]

2. SARIMA Model

The Seasonal ARIMA (SARIMA) model extends ARIMA to include seasonal components:

(a) Seasonal Autoregressive (SAR) Component

$$X_t = c + \sum_{j=1}^{P} \phi_j X_{t-j} + \sum_{k=1}^{p} \theta_k \epsilon_{t-k} + \epsilon_t \tag{2.52}$$

where P is the seasonal autoregressive order, ϕ_j are the seasonal autoregressive coefficients, and ϵ_t are white noise error terms.

(b) Seasonal Integrated (SI) Component

$$Y_t = (1 - B^s)^D (1 - B)^d X_t \tag{2.53}$$

where s is the seasonal period, D is the order of seasonal differencing, d is the order of non-seasonal differencing, and B^s and B are seasonal and non-seasonal backshift operators, respectively.

(c) Seasonal Moving Average (SMA) Component

$$X_t = \mu + \sum_{j=1}^{Q} \theta_j \epsilon_{t-j} + \sum_{k=1}^{q} \phi_k \epsilon_{t-k} \tag{2.54}$$

where Q is the seasonal moving average order, θ_j are the seasonal moving average coefficients, and ϵ_t are white noise error terms.[38]

In conclusion, ARIMA and SARIMA models are powerful tools for time series analysis and forecasting:

- **ARIMA:** Models non-seasonal time series data using autoregressive, integrated, and moving average components.
- **SARIMA:** Extends ARIMA to include seasonal variations, incorporating seasonal autoregressive, integrated, and moving average components.
- Both models require selection of parameters p, d, q for ARIMA and P, D, Q, s for SARIMA, often determined through model diagnostics such as ACF and PACF plots.
- Effective forecasting with ARIMA and SARIMA depends on the stationarity of the data and correct specification of model parameters.

2.3.3 Deep Learning Methods

Time series analysis plays a crucial role in various fields such as finance, healthcare, and climate modeling. Predicting future trends based on historical data is a challenging task that often requires models capable of capturing complex temporal dependencies. Traditional recurrent neural networks (RNNs) and Long Short-Term Memory networks (LSTMs) have been widely used for such tasks, but they can suffer from limitations such as sequential computation and difficulty in capturing long-term dependencies.

Temporal Convolutional Networks (TCNs) offer a promising alternative by leveraging one-dimensional convolutions, residual connections, and dilated convolutions to effectively model sequential data. This part explores the foundational principles, architectural components, and mathematical formulations of TCNs, highlighting their advantages and applications in time series analysis.[39]

1. One-Dimensional Convolutional Layers

TCNs begin with one-dimensional convolutional layers that operate across the temporal dimension of the input sequence $x \in \mathbb{R}^{T \times C}$, where T is the sequence length and C denotes the number of features.

$$y[t, f] = \sum_{c=1}^{C} \sum_{i=1}^{k} W[i, c, f] \cdot x[t + i - \lfloor k/2 \rfloor, c] \tag{2.55}$$

Here, $W \in \mathbb{R}^{k \times C \times F}$ represents the convolutional filter with kernel size k, and F denotes the number of output channels.

2. Residual Blocks

To facilitate the training of deeper networks, TCNs incorporate residual blocks. These blocks employ skip connections to alleviate the vanishing gradient problem and enable effective optimization of deep architectures:

$$h^{(l+1)} = \text{ReLU}(h^{(l)} + \mathcal{F}(h^{(l)}, W^{(l)})) \tag{2.56}$$

where $h^{(l)}$ is the output of the previous layer, $\mathcal{F}$ denotes the residual function typically consisting of convolutional layers, and $W^{(l)}$ are the parameters associated with the layers within the block.

3. Dilated Convolutions

Dilated convolutions play a crucial role in TCNs by increasing the receptive field without a proportional increase in parameters. For a dilation factor d, the output $y[t]$ is computed as:

$$y[t] = \sum_{i=1}^{k} W[i] \cdot x[t + d \cdot i] \tag{2.57}$$

This allows TCNs to capture long-range dependencies in the input sequence efficiently.

The overall output $\hat{y}$ of a TCN can be expressed as:

$$\hat{y} = \text{softmax}(W^{(out)} \cdot h^{(L)} + b^{(out)}) \tag{2.58}$$

where $h^{(L)}$ represents the output after the final layers (including residual blocks), $W^{(out)}$ and $b^{(out)}$ are the parameters of the output layer, and softmax is the activation function for classification tasks.

In conclusion, Temporal Convolutional Networks (TCNs) offer a compelling alternative to traditional recurrent architectures for time series analysis. By leveraging one-dimensional convolutions, residual connections, and dilated convolutions,

TCNs can effectively capture temporal dependencies and achieve state-of-the-art performance in various sequential data tasks. Understanding the architectural components and mathematical formulations of TCNs is crucial for leveraging their capabilities in practical applications such as forecasting, anomaly detection, and signal processing.

2.4 Sensor Fusion

2.4.1 Bayesian Filters

Bayesian filters are a class of state estimation methods based on Bayesian theorem, including Kalman Filters and their nonlinear extensions such as Extended Kalman Filters (EKF) and Unscented Kalman Filters (UKF). These methods fuse sensor measurements with the system's dynamic model to generate an optimal estimate of the system state.

1. Kalman Filters

Kalman Filters are designed for linear dynamic systems with Gaussian noise.[40] There are two key phases:

(a) Prediction Phase

$$\hat{x}_k^- = F_k \hat{x}_{k-1} + B_k u_k \tag{2.59}$$

$$P_k^- = F_k P_{k-1} F_k^\top + Q_k \tag{2.60}$$

(b) Update Phase

$$K_k = P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1} \tag{2.61}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H_k \hat{x}_k^-) \tag{2.62}$$

$$P_k = (I - K_k H_k) P_k^- \tag{2.63}$$

where:

- $\hat{x}_k$ is the state estimate at time k,
- P_k is the state covariance matrix,
- F_k is the state transition matrix,
- B_k is the control input matrix,
- u_k is the control input vector,
- Q_k is the process noise covariance matrix,
- H_k is the measurement matrix,

- R_k is the measurement noise covariance matrix,
- z_k is the measurement vector,
- K_k is the Kalman gain.

2. Extended Kalman Filters (EKF)

EKF extends Kalman Filters to handle nonlinear dynamics by linearizing the system model around the current state estimate. There are two key phases:

(a) Prediction Phase

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_k) \tag{2.64}$$

$$P_k^- = F_k P_{k-1} F_k^\top + Q_k \tag{2.65}$$

2.4.1.1 (b) Update Phase

$$K_k = P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1} \tag{2.66}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-)) \tag{2.67}$$

$$P_k = (I - K_k H_k) P_k^- \tag{2.68}$$

where f and h represent the nonlinear process and measurement models, respectively, and F_k is the Jacobian matrix of $f(\cdot)$.[41]

3. Unscented Kalman Filters (UKF)

UKF avoids linearization by selecting a minimal set of sample points (sigma points) to represent the distribution of the state estimate and covariance. There are two key steps:

(a) Sigma Point Calculation

$$X_k^{(i)} = \hat{x}_k + [\sqrt{(n+\lambda)P_k}]_i, \quad i = 0, \ldots, 2n \tag{2.69}$$

(b) Prediction and Update

$$\hat{x}_k^- = \sum_{i=0}^{2n} W_m^{(i)} f(X_k^{(i)}, u_k) \tag{2.70}$$

$$P_k^- = \sum_{i=0}^{2n} W_c^{(i)} [f(X_k^{(i)}, u_k) - \hat{x}_k^-][f(X_k^{(i)}, u_k) - \hat{x}_k^-]^\top + Q_k \tag{2.71}$$

$$K_k = P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1} \tag{2.72}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-)) \tag{2.73}$$

$$P_k = P_k^- - K_k H_k P_k^- \tag{2.74}$$

where $W_m^{(i)}$ and $W_c^{(i)}$ are weights associated with the sigma points, and f and h are the process and measurement functions, respectively.[42]

Bayesian filters, including Kalman Filters, Extended Kalman Filters (EKF), and Unscented Kalman Filters (UKF), are indispensable tools for state estimation in dynamic systems. They leverage Bayesian principles to fuse prior knowledge with real-time sensor measurements, providing optimal state estimates even in the presence of noise and nonlinearities. These filters find applications in various domains, including robotics, navigation, and signal processing, where accurate state estimation is crucial for decision-making and control. As technology advances, Bayesian filters continue to evolve, adapting to increasingly complex systems and environments, thereby remaining at the forefront of modern estimation techniques.

2.4.2 Particle Filters

Particle filters, or Sequential Monte Carlo (SMC) methods, are pivotal for state estimation in nonlinear and non-Gaussian systems.[43, 44] They approximate the posterior distribution of the system state using a set of weighted particles, making them robust in scenarios where traditional methods like Kalman filters are inadequate.

At each time step t, particle filters maintain an ensemble of particles $\{\mathbf{x}_t^{(i)}, w_t^{(i)}\}_{i=1}^N$, where $\mathbf{x}_t^{(i)}$ represents a hypothesized state and $w_t^{(i)}$ its associated weight. The algorithm follows these key steps:

1. **Prediction**:
$$\mathbf{x}_t^{(i)} \sim p(\mathbf{x}_t|\mathbf{x}_{t-1}^{(i)}) \tag{2.75}$$
Here, $p(\mathbf{x}_t|\mathbf{x}_{t-1}^{(i)})$ denotes the transition model predicting the state at time t given the previous state $\mathbf{x}_{t-1}^{(i)}$.

2. **Weight Update**:
$$w_t^{(i)} \propto p(\mathbf{z}_t|\mathbf{x}_t^{(i)}) \cdot w_{t-1}^{(i)} \tag{2.76}$$
The weight $w_t^{(i)}$ is updated based on the likelihood $p(\mathbf{z}_t|\mathbf{x}_t^{(i)})$, where $\mathbf{z}_t$ is the observation at time t.

3. **Resampling**:
$$\text{Resample particles based on } \{w_t^{(i)}\}$$
Resampling ensures that particles with higher weights are replicated, while those with lower weights are discarded, maintaining diversity in the particle set.

Particle filters offer several advantages over traditional estimation methods:

- **Nonlinearity and Non-Gaussianity**: Particle filters excel in estimating states where the dynamics are nonlinear and the distributions are non-Gaussian.
- **Flexibility**: They can accommodate various system models without requiring explicit knowledge of their exact forms, relying instead on the ability to sample from these models.

Particle filters, despite their effectiveness, also have certain limitations that must be considered:

- **Computational Intensity**: The computational complexity scales with the number of particles N, which can be demanding for large state spaces or real-time applications.
- **Particle Degeneracy**: Without efficient resampling strategies, particle diversity diminishes over time, reducing the effectiveness of the filter.

Particle filters use Monte Carlo methods to approximate complex posterior distributions in dynamic systems, crucial for state estimation in fields like robotics and econometrics. Despite challenges such as computational intensity and particle degeneracy, they offer robustness and flexibility. Ongoing research in optimization and adaptive resampling aims to improve their real-world applicability, making them pivotal in probabilistic inference beyond linear Gaussian models.

2.4.3 Information Fusion

Information fusion is a critical process in multisensor systems aimed at integrating data from multiple sources to enhance system performance. This part explores the mathematical foundations and theoretical frameworks underlying information fusion.[45]

Data preprocessing is the initial step where raw sensor measurements $\mathbf{z}_i$ are transformed into a standardized format $\mathbf{z}'_i$ suitable for further analysis. This often includes normalization to ensure all sensor data are on a consistent scale:

$$\mathbf{z}'_i = f_{\text{norm}}(\mathbf{z}_i) \tag{2.77}$$

where f_{norm} represents the normalization function.

Feature extraction aims to capture relevant information from preprocessed sensor data $\mathbf{z}'_i$. Techniques like Principal Component Analysis (PCA) are commonly used to reduce dimensionality and highlight meaningful patterns:

$$\mathbf{f}_i = \mathbf{V}_i^T(\mathbf{z}'_i - \mathbf{m}_i) \tag{2.78}$$

where $\mathbf{V}_i$ are eigenvectors and $\mathbf{m}_i$ is the mean vector.

Feature fusion combines the extracted features $\mathbf{f}_i$ from all sensors into a unified feature vector $\mathbf{f}_{\text{fused}}$ that captures complementary information from multiple sources:

$$\mathbf{f}_{\text{fused}} = f_{\text{fusion}}(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_N) \tag{2.79}$$

This step enhances the discriminative power and robustness of the system.

The fused feature vector $\mathbf{f}_{\text{fused}}$ serves as input for decision-making processes, where various algorithms or models utilize this consolidated information to make informed decisions:

$$\hat{y} = f_{\text{decision}}(\mathbf{f}_{\text{fused}}) \tag{2.80}$$

These decisions are crucial in applications ranging from autonomous systems to healthcare monitoring.

Information fusion leverages various theoretical frameworks to integrate diverse sensor data effectively. Here, I explore three fundamental approaches: probabilistic fusion, fuzzy set theory, and neural networks.

1. Probabilistic Fusion

Probabilistic fusion employs Bayesian inference to combine sensor data probabilistically, accommodating uncertainties:

$$P(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N|\mathbf{y}) = \frac{P(\mathbf{y}|\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N)P(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N)}{P(\mathbf{y})} \tag{2.81}$$

2. Fuzzy Set Theory

Fuzzy set theory handles imprecise or vague data through fuzzy logic operations, providing a flexible framework for managing uncertain sensor inputs:

$$\mu_{\text{fusion}}(x) = \sup_{x_1, x_2, \ldots, x_N} \min(\mu_1(x_1), \mu_2(x_2), \ldots, \mu_N(x_N)) \tag{2.82}$$

3. Neural Networks

Neural networks learn complex relationships and patterns from fused feature vectors, offering adaptability and scalability in dynamic environments:

$$\mathbf{f}_{\text{fused}} = f_{\text{NN}}(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_N) \tag{2.83}$$

Information fusion optimizes multi-sensor systems by integrating diverse sensor data through systematic pre-processing, feature extraction, and fusion. Supported by theoretical frameworks like probabilistic reasoning, fuzzy set theory, and neural networks, it enhances system performance for decision-making across various domains. By seamlessly connecting each stage, information fusion ensures that multi-sensor systems operate effectively in capturing and interpreting complex real-world data, thereby facilitating advancements in fields such as remote sensing, surveillance, and autonomous systems.

2.4.4 Deep Learning based Sensor Fusion

In sensor fusion, assuming multiple sensors measure the same physical quantity x, the measurement from sensor i can be expressed as:

$$z_i = x + \epsilon_i$$

where $\epsilon_i \sim \mathcal{N}(0, R_i)$ represents the measurement noise of sensor i, and R_i is its covariance matrix. The goal is to estimate the optimal value $\hat{x}$ of the state x using measurements from all sensors $z = [z_1, z_2, \ldots, z_m]^T$.[46]

Using the Minimum Mean Squared Error (MMSE) criterion, the optimal fusion estimate $\hat{x}$ is given by:

$$\hat{x} = (R^{-1}z)/(R^{-1}\mathbf{1})$$

Here, $R = \text{diag}(R_1, R_2, \ldots, R_m)$ represents the covariance matrix of measurement errors, and $\mathbf{1}$ is a vector of ones. This formula is derived from solving a linear least squares problem for state estimation, taking into account the weights of each sensor measurement and their respective measurement errors.

The covariance P of the optimal estimate $\hat{x}$ can be computed as:

$$P = (R^{-1} + \Lambda)^{-1}$$

where $\Lambda = \lambda I$, λ is a small positive number (typically used for numerical stability), and I is the identity matrix. This equation describes the covariance of estimation errors, considering both measurement errors and uncertainties in the estimation process.

Deep learning models for sensor fusion primarily utilize Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). These architectures automatically learn complex feature representations and extract key information from extensive datasets, thereby enhancing the accuracy and robustness of state estimation.

- **Convolutional Neural Networks (CNNs)**: CNNs are effective for processing spatial information, especially for tasks involving image data.[47] Through convolutional and pooling layers, CNNs extract spatial features from images and use fully connected layers for state estimation.

$$\hat{x} = \text{CNN}(I; \theta) \tag{2.84}$$

- **Recurrent Neural Networks (RNNs)**: RNNs are suitable for processing temporal information, such as time-series data or streams from sensors.[48] By leveraging recurrent connections, RNNs capture temporal dependencies in data and continuously model state changes.

$$h_t = \text{RNN}(z_t, h_{t-1}; \theta) \tag{2.85}$$

$$\hat{x} = \text{OutputLayer}(h_T; \theta) \tag{2.86}$$

  Here, h_t denotes the hidden state, and OutputLayer represents the output layer of the network with parameters θ.

These neural network architectures can be applied not only for individual sensor data processing but also for integrating multi-sensor data in an end-to-end learning and optimization framework.

To optimize the deep learning models for accurate state estimation, Mean Squared Error (MSE) is commonly used as the loss function. During training, the gradients of the loss function with respect to the network parameters θ are computed using

backpropagation, and an optimizer (such as stochastic gradient descent) updates the parameters to minimize the loss function:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}(\theta) \tag{2.87}$$

where η is the learning rate that controls the step size of parameter updates.

Deep learning demonstrates significant capabilities in sensor fusion by learning intricate feature representations and optimizing performance through extensive data analysis. This technology not only enhances the accuracy of environmental perception but also strengthens adaptability to complex environmental changes, offering new avenues and applications for the development of intelligent sensing systems.

Chapter 3

Methods

In this section, I delineate the methodologies employed in my study. For outlier detection, I applied the autoregressive integrated moving average (ARIMA) method. In sensor calibration, I utilized a sophisticated deep learning-based approach known as Deep Calibration Method (DeepCM). For short-term forecasting tasks, Long Short-Term Memory (LSTM) networks were employed due to their capability to capture temporal dependencies in sequential data. Additionally, in sensor fusion tasks, Gaussian Process Regression (GPR) was leveraged for its ability to model complex relationships and uncertainties in multidimensional data.

3.1 ARIMA Model for Outlier Detection

The ARIMA (AutoRegressive Integrated Moving Average) model, a classical tool in time series analysis, is not only employed for forecasting and model fitting but also proves effective in outlier detection. Outlier detection holds significant importance in time series analysis as it aids in identifying data points that deviate significantly from the expected pattern.

3.1.1 ARIMA Model Construction and Parameter Estimation

To build an ARIMA model, we first ensure the time series Y_t is stationary. A stationary time series has constant mean and variance over time. If the series is not stationary, we apply differencing. The differencing operator ∇ is defined as $\nabla Y_t = Y_t - Y_{t-1}$. Higher order differences can be computed as follows:

$$\nabla^d Y_t = (1 - B)^d Y_t \tag{3.1}$$

where B is the backshift operator, such that $BY_t = Y_{t-1}$.[49]

For example:

$$\nabla Y_t = Y_t - Y_{t-1} \tag{3.2}$$

$$\nabla^2 Y_t = \nabla(\nabla Y_t) = (1 - B)(1 - B)Y_t = Y_t - 2Y_{t-1} + Y_{t-2} \tag{3.3}$$

To identify the AR and MA components, we examine the autocorrelation function (ACF) and partial autocorrelation function (PACF) plots:

- **ACF**: The autocorrelation function $\rho(k)$ measures the correlation between Y_t and Y_{t-k}:

$$\rho(k) = \frac{\gamma(k)}{\gamma(0)} \tag{3.4}$$

  where $\gamma(k)$ is the autocovariance function:

$$\gamma(k) = \mathbb{E}[(Y_t - \mu)(Y_{t-k} - \mu)] \tag{3.5}$$

- **PACF**: The partial autocorrelation function $\phi(k)$ measures the direct correlation between Y_t and Y_{t-k}, removing the effect of intermediate lags.

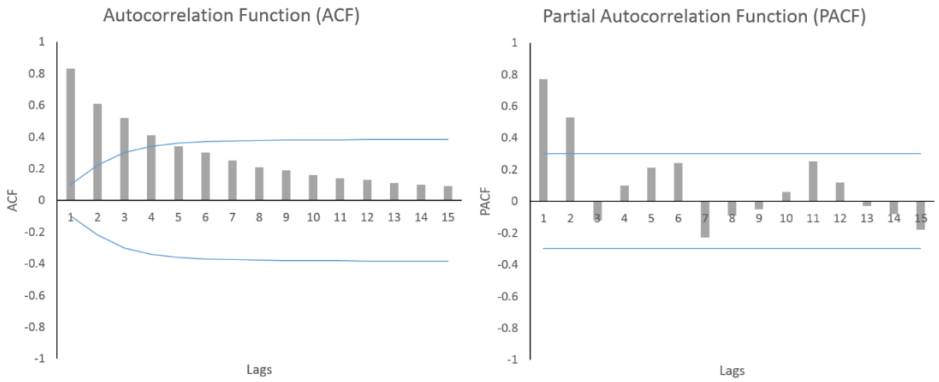


Figure 3.1.1: ACF and PACF of AR Process

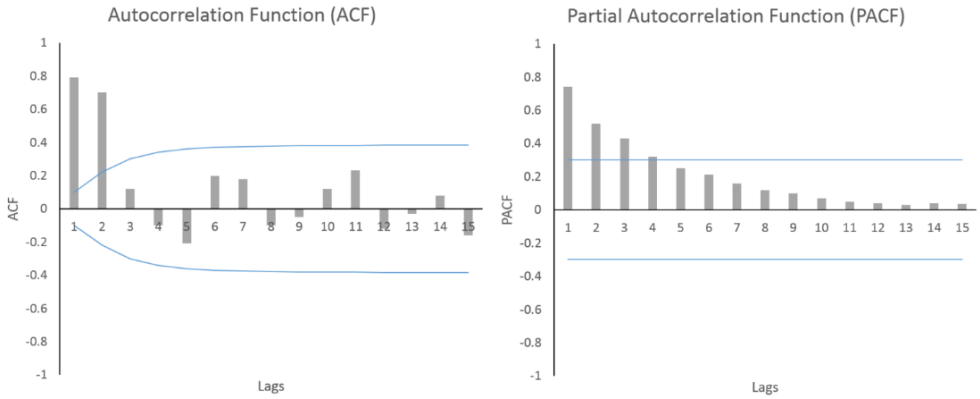


Figure 3.1.2: ACF and PACF of MA Process

Based on the ACF and PACF plots in figure 3.1.1, figure 3.1.2 and figure 3.1.3, we identify the orders of the AR (p) and MA (q) components:

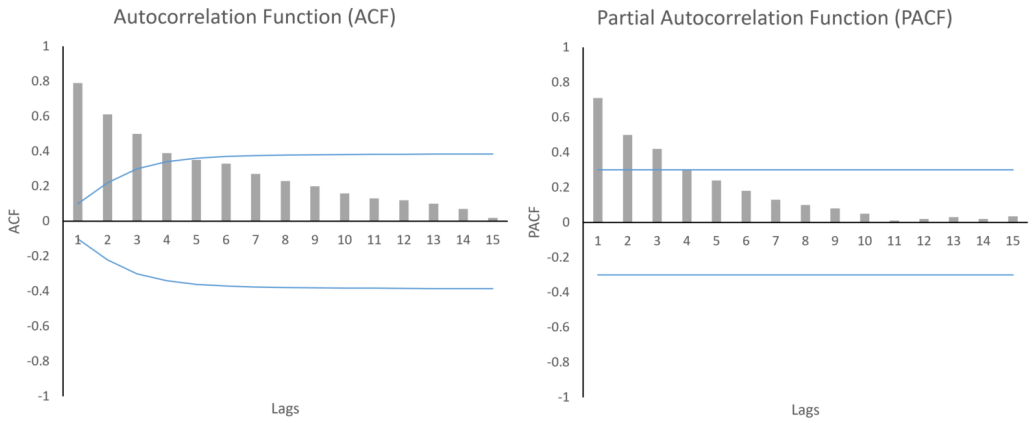


Figure 3.1.3: ACF and PACF of ARMA Process

- **AR (p)**: The autoregressive part of the model is given by:

$$Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \ldots + \phi_p Y_{t-p} + \epsilon_t \tag{3.6}$$

  where ϕ_i are the AR parameters and ϵ_t is the white noise error term.

- **MA (q)**: The moving average part of the model is given by:

$$Y_t = \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} \tag{3.7}$$

  where θ_i are the MA parameters.

Combining the differencing, AR, and MA components, we get the ARIMA model:

$$\nabla^d Y_t = \phi(B)\epsilon_t \tag{3.8}$$

where $\phi(B)$ is the AR polynomial:

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p \tag{3.9}$$

The full ARIMA model is then:

$$(1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p)(1 - B)^d Y_t = (1 + \theta_1 B + \theta_2 B^2 + \ldots + \theta_q B^q)\epsilon_t \tag{3.10}$$

Constructing an ARIMA model involves initial checks for stationarity, ACF and PACF analysis to identify p and q.[50] This systematic approach ensures the selected model accurately represents the time series data, enabling reliable forecasting and analysis.

3.1.2 Residual Calculation

The process of calculating residuals in ARIMA (AutoRegressive Integrated Moving Average) modeling involves several mathematical steps to evaluate how well the model captures the patterns inherent in a time series.[51]

To begin, I outlined the essential steps involved in fitting an ARIMA model to a time series and calculating residuals.

1. Model Fitting

- **Model Specification**: Define the ARIMA model based on the time series characteristics, denoted as ARIMA(p, d, q):

$$(1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p)(1 - B)^d y_t = (1 + \theta_1 B + \theta_2 B^2 + \cdots + \theta_q B^q)\epsilon_t \quad (3.11)$$

  where B is the backshift operator, y_t is the observed value at time t, ϵ_t is the white noise error term, and ϕ_i, θ_i are the autoregressive and moving average coefficients, respectively.

- **Parameter Estimation**: Estimate model parameters ($\{\phi_i\}$, $\{\theta_i\}$, σ^2) using methods such as maximum likelihood estimation.

2. Prediction Generation

- **In-Sample Prediction**: Use the fitted ARIMA model to generate in-sample predictions $\{\hat{y}_t\}$ for each observation y_t in the time series:

$$\hat{y}_t = \mu + \sum_{i=1}^{p} \phi_i y_{t-i} + \epsilon_t - \sum_{j=1}^{q} \theta_j \epsilon_{t-j} \quad (3.12)$$

  where μ is the constant term.

3. Residual Computation

- **Compute Residuals**: Calculate residuals e_t by subtracting each observed value y_t from its corresponding predicted value $\hat{y}_t$:

$$e_t = y_t - \hat{y}_t \quad (3.13)$$

  This provides a measure of the difference between the actual observation and the model's prediction.

The calculation and analysis of residuals in ARIMA modeling are essential for validating the model's performance and identifying areas for improvement. By systematically assessing discrepancies between actual observations and model predictions, practitioners can enhance the model's ability to forecast future values accurately.

3.1.3 Outlier Detection

Utilize residuals for outlier detection. A common approach involves computing standardized residuals (e.g., residuals divided by their standard deviation) and comparing them against predefined thresholds. Larger standardized residuals typically correspond to outlier observations, indicating significant deviations from the model's expected behavior.

1. Standardized Residuals Calculation

Given observed values y_i and predicted values $\hat{y}_i$ from a statistical model, the residual e_i for the i-th observation is:

$$e_i = y_i - \hat{y}_i \tag{3.14}$$

To standardize the residual e_i, divide it by an estimate of its standard deviation $\hat{\sigma}_i$:

$$r_i = \frac{e_i}{\hat{\sigma}_i} \tag{3.15}$$

where $\hat{\sigma}_i$ can be estimated using methods such as the root mean squared error (RMSE) or the standard error of the residuals.

2. Estimating Standard Deviation $\hat{\sigma}_i$

The RMSE (Root Mean Squared Error) is given by:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} e_i^2} \tag{3.16}$$

Alternatively, the standard error of the residuals $\hat{\sigma}_i$ can be estimated as:

$$\hat{\sigma}_i = \sqrt{\frac{\sum_{i=1}^{n} e_i^2}{n-p}} \tag{3.17}$$

where n is the number of observations and p is the number of parameters in the model.

To identify outliers, compare the absolute value of each standardized residual $|r_i|$ against a predefined threshold λ. Observations with $|r_i| > \lambda$ are flagged as potential outliers.

The use of standardized residuals for outlier detection is grounded in statistical theory:

- **Normalization:** Standardizing residuals ensures they are on a comparable scale, facilitating meaningful comparisons.
- **Distributional Assumption:** Under normal distribution assumptions, standardized residuals r_i follow a standard normal distribution $N(0,1)$, enabling statistical tests for outlier detection.

- **Model Assessment:** Outliers in residuals often indicate model inadequacies, such as nonlinearity or influential data points.

By computing and analyzing standardized residuals, analysts can effectively detect outliers that may distort statistical models, enhancing the robustness and accuracy of data analysis.

3.1.4 Threshold Selection

Selecting appropriate thresholds for outlier detection is crucial in statistical analysis and machine learning. This section explores methods based on residual analysis, particularly focusing on the standard deviation method and the interquartile range (IQR) method.

1. Standard Deviation Method

The standard deviation method utilizes the variability of residuals around the mean to set outlier thresholds. Let e_i represent the residual for the i-th observation, and σ_e denote the standard deviation of residuals. Outliers can be defined as:

$$|e_i| > k \cdot \sigma_e$$

where k is a multiplier determined based on the desired level of sensitivity. Typically, k is chosen such that it captures observations significantly deviating from the mean, assuming a normal distribution of residuals.

2. Interquartile Range (IQR) Method

The IQR method is robust against outliers and works well with skewed distributions. The IQR is defined as $\text{IQR} = Q3 - Q1$, where $Q1$ and $Q3$ represent the 25th and 75th percentiles of the residuals, respectively.[52] Outliers can be identified using:

$$e_i < Q1 - k \cdot \text{IQR} \quad \text{or} \quad e_i > Q3 + k \cdot \text{IQR}$$

where k is typically set to 1.5 or 3, depending on the application's sensitivity to outliers.

The choice of threshold depends significantly on the specific context and objectives of the analysis:

- **Financial Fraud Detection**: In this context, thresholds are often conservative to minimize false positives. For instance, setting $k = 3$ in the standard deviation method ensures that only extreme anomalies are flagged as outliers, reducing the risk of overlooking fraudulent activities.
- **Sensor Data Monitoring**: Here, thresholds may be set more liberally to detect significant anomalies promptly. A lower k value in the standard deviation method (e.g., $k = 2$) allows for quicker detection of deviations from expected sensor readings, facilitating timely responses to potential equipment malfunctions or environmental changes.

In conclusion, selecting thresholds for outlier detection involves balancing statistical rigor with contextual understanding. Methods such as the standard deviation and IQR offer flexible approaches depending on the distributional characteristics of residuals and the specific requirements of the application. By carefully choosing and validating thresholds, practitioners can achieve robust and effective outlier detection tailored to their data and analytical goals.

3.1.5 Evaluation

Assessing the performance of outlier detection methods in time series data is crucial for identifying data points that deviate significantly from expected patterns. Various evaluation metrics, including the ROC curve, provide insights into the effectiveness of these methods across different thresholds.[53]

The ROC curve (Receiver Operating Characteristic curve) plots the False Positive Rate (FPR) against the True Positive Rate (TPR), defined as:

$$\text{TPR} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3.18}$$

$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} \tag{3.19}$$

Outlier detection in time series identifies data points significantly deviating from expected patterns. ROC curves help evaluate methods across different outlier score thresholds.

Evaluation Steps

- **Data Preparation**: Divide data into training and testing sets. Train the model on training data and evaluate on test data.
- **Outlier Detection**: Compute outlier scores o_i for each time step i. Define outliers based on a threshold τ:

$$\text{Outlier at } i = \begin{cases} 1 & \text{if } o_i \geq \tau \\ 0 & \text{otherwise} \end{cases} \tag{3.20}$$

- **Construct ROC Curve**: Vary τ and calculate TPR and FPR:

$$\text{TPR}(\tau) = \frac{\text{TP}(\tau)}{\text{TP}(\tau) + \text{FN}(\tau)} \tag{3.21}$$

$$\text{FPR}(\tau) = \frac{\text{FP}(\tau)}{\text{FP}(\tau) + \text{TN}(\tau)} \tag{3.22}$$

  where TP, FN, FP, TN represent true positives, false negatives, false positives, and true negatives, respectively.
- **Calculate AUC**: Compute the Area Under the ROC Curve (AUC) to quantify performance.

Considerations

- ROC curves provide a robust evaluation method but require considering temporal dependencies and data correlations in time series.
- They facilitate comparison and parameter tuning to improve outlier detection accuracy.

Using ROC curves with outlier detection methods offers a systematic approach to assess and optimize performance. It guides threshold selection and method refinement, ensuring effective differentiation between normal and anomalous behaviors in time series data. ROC analysis supports informed decision-making in enhancing outlier detection systems across various applications.

In summary, the ARIMA model's outlier detection method combines the advantages of time series model fitting and residual analysis. It represents a theoretically sound approach widely validated in practical applications for identifying outliers in time series data. This method effectively identifies data points that deviate significantly from the expected pattern, facilitating more accurate data analysis and decision-making processes.

3.2 Deep Calibration Method (DeepCM) for Sensor Calibration

The calibration of low-cost air monitoring sensors is crucial for ensuring accurate and reliable environmental data collection. The DeepCM (Deep Calibration Model) provides an advanced solution leveraging deep learning techniques to enhance the precision of sensor readings. This guide outlines the comprehensive steps involved in utilizing the DeepCM model for sensor calibration, from data preparation to model training and application.

3.2.1 Data Preparation

1. **Data Collection**:
    - Collect raw data from low-cost sensors along with high-accuracy reference data.
    - Include features such as timestamps, sensor readings, and environmental conditions like temperature, humidity, etc.
2. **Data Pre-processing**:
    - **Data Cleaning**: Handle missing values, outliers, and noise in the data.
    - **Data Alignment**: Ensure alignment between sensor data and reference data timestamps for consistent records.
    - **Feature Engineering**: Transform timestamps into time series and create features for local, global, and periodic characteristics.

3.2.2 Building the DeepCM Model

Recurrent Neural Networks (RNNs) are a type of recursive neural network designed to handle sequential data. Their structure allows information to propagate across time steps, utilizing hidden states to retain and update historical information.[54]

The core update of the hidden state $\mathbf{h}_t$ in an RNN is given by:

$$\mathbf{h}_t = \phi(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \tag{3.23}$$

where:

- $\mathbf{x}_t$ is the input vector at time step t,
- $\mathbf{h}_{t-1}$ is the hidden state from the previous time step,
- $\mathbf{W}_{hh}$ is the weight matrix for the recurrent connections,
- $\mathbf{W}_{xh}$ is the weight matrix for the input connections,
- $\mathbf{b}_h$ is the bias vector,
- ϕ is the activation function applied element-wise (e.g., tanh).

The output $\hat{\mathbf{y}}_t$ at time step t is produced by a fully connected layer followed by a softmax activation function:

$$\mathbf{z}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y \tag{3.24}$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{z}_t) \tag{3.25}$$

where:

- $\mathbf{W}_{hy}$ is the weight matrix connecting the hidden state to the output,
- $\mathbf{b}_y$ is the bias vector,
- $\mathbf{z}_t$ is the logit vector before applying softmax,
- softmax is the softmax function, which normalizes the logits to obtain probabilities.

The specific workflow of RNN is shown in the figure 3.2.1.

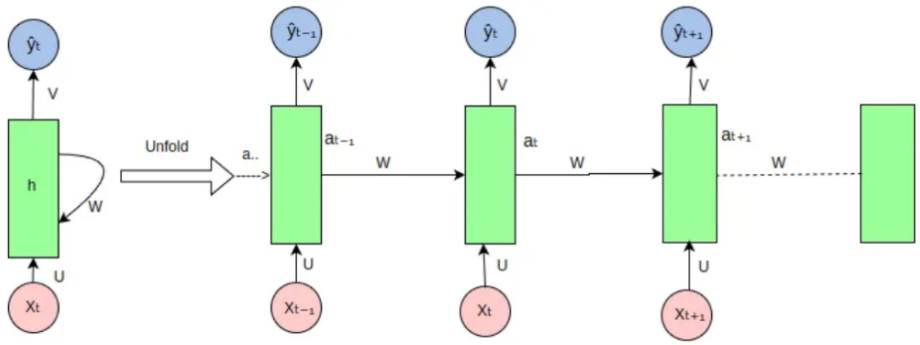


Figure 3.2.1: RNN Workflow

RNNs possess several key characteristics that make them suitable for handling time series data:

- **Sequence Dependency**: RNNs capture dependencies over time due to their recurrent connections. This allows them to model sequences where each element depends on previous ones, making them effective for tasks requiring memory over time.
- **Parameter Sharing**: Parameters in RNNs (specifically $\mathbf{W}_{hh}$ and $\mathbf{W}_{xh}$) are shared across time steps. This shared structure enables the model to generalize across sequences of varying lengths and learn from sequential patterns efficiently.

In summary, RNNs leverage their recurrent structure to maintain and update hidden states over time, allowing them to effectively model and predict sequential data. Their parameter sharing mechanism optimizes the use of trainable parameters.

3.2.3 Training the DeepCM Model

To train the RNN model, a suitable loss function is defined to measure the difference between predicted and actual observations.

A commonly used loss function is the Mean Squared Error (MSE), calculated as:

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \|\hat{\mathbf{y}}_t - \mathbf{y}_t\|_2^2 \tag{3.26}$$

Here, T denotes the sequence length, $\hat{\mathbf{y}}_t$ represents the model's prediction at time step t, and $\mathbf{y}_t$ is the corresponding ground truth observation.

Minimizing the loss function $\mathcal{L}$ adjusts model parameters to make predicted outputs closely match actual observations.

Training Recurrent Neural Networks (RNNs) involves optimizing a loss function to adjust model parameters. Stochastic Gradient Descent (SGD) and its advanced variants, such as the Adam optimizer, are commonly used for this purpose.

1. Stochastic Gradient Descent (SGD)

SGD is a fundamental optimization algorithm used to minimize the loss function $\mathcal{L}$ by iteratively updating model parameters θ:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) \tag{3.27}$$

where:

- η is the learning rate, controlling the size of the step taken towards the minimum.
- $\nabla_\theta \mathcal{L}(\theta_t)$ denotes the gradient of the loss function with respect to parameters θ at iteration t.

SGD computes gradients based on a mini-batch of training data at each iteration, making it computationally efficient but potentially noisy.[55]

2. Adam Optimizer

Adam optimizer combines the advantages of both AdaGrad and RMSProp algorithms. It maintains adaptive learning rates for each parameter and stores exponentially decaying averages of past gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta \mathcal{L}(\theta_t), \tag{3.28}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta \mathcal{L}(\theta_t))^2, \tag{3.29}$$

where:

- m_t and v_t are estimates of the first moment (mean) and second moment (uncentered variance) of the gradients, respectively.
- β_1 and β_2 are exponential decay rates for the moment estimates (typically $\beta_1 = 0.9$ and $\beta_2 = 0.999$).

The Adam update rule adjusts each parameter θ as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} \cdot m_t \tag{3.30}$$

where ϵ is a small constant to prevent division by zero.[56]

Adam is advantageous due to its adaptive nature, which adjusts learning rates for each parameter individually based on the magnitude of recent gradients and the decay rates of moment estimates. This makes Adam well-suited for training RNNs on complex datasets with non-stationary and sparse gradients.

Optimization algorithms like SGD and Adam are crucial for training RNN models effectively on sensor data. They enable the model to learn from historical information, adjust parameters efficiently, and improve prediction performance over time.

3.2.4 Applying the Model for Sensor Calibration

In this section, we outline the application of the DeepCM model for calibrating sensor data:

1. **Input New Data**:
   - Feed new, uncalibrated sensor data $\mathbf{X}_{\text{raw}}$ into the DeepCM model, which consists of time-series observations.
2. **Model Inference**:
   - The DeepCM model, leveraging learned parameters $\boldsymbol{\theta}$, processes $\mathbf{X}_{\text{raw}}$ to produce calibrated outputs $\mathbf{Y}_{\text{calibrated}}$:

     $$\mathbf{Y}_{\text{calibrated}} = f_{\boldsymbol{\theta}}(\mathbf{X}_{\text{raw}})$$

     Here, $f_{\boldsymbol{\theta}}$ represents the model's transformation function.

3. **Output Results**:
    - Obtain calibrated sensor readings $\mathbf{Y}_{\text{calibrated}}$, providing more accurate estimates of pollutant concentrations or other environmental parameters. These refined measurements are crucial for informed decision-making in environmental monitoring.

The DeepCM model's ability to learn temporal patterns and correct biases in sensor data enhances the reliability of environmental monitoring systems, contributing to effective environmental management strategies.

3.2.5 Model Evaluation and Fine-Tuning

Effective evaluation and refinement are crucial stages in developing robust predictive models.

1. **Evaluate Model Performance**:
    - Model effectiveness is assessed by comparing sensor readings with high-accuracy reference data. This comparison quantifies the model's accuracy and reliability in real-world scenarios.
    - Evaluation metrics, such as Mean Squared Error (MSE), are commonly used. MSE is defined as:

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

      where $\hat{y}_i$ is the predicted value and y_i is the actual value. Lower MSE values indicate better model performance.

2. **Refine the Model**:
    - Adjustments are made based on evaluation results to optimize the model. This includes tuning hyper-parameters, increasing data samples, and improving data pre-processing steps.
    - Iterative refinement cycles ensure continual improvement of models like DeepCM, aligning predictions more closely with real-world data.

In summary, model evaluation and fine-tuning iteratively enhance predictive accuracy and reliability, crucial for applications such as the DeepCM model.

3.3 LSTM for Short-term Forecasting

In this section, I will introduce three Long Short-Term Memory (LSTM) based short-time prediction methods, namely the single-sensor LSTM model, the single-sensor LSTM encoder-decoder model, and the multi-sensor LSTM encoder-decoder model.

3.3.1 Single Sensor LSTM Model

3.3.1.1 LSTM Structure

LSTM networks are designed to handle the vanishing gradient problem in traditional RNNs by utilizing specialized gates that control the flow of information over time. The specific structure of LSTM is shown in the figure 3.3.1.[57]

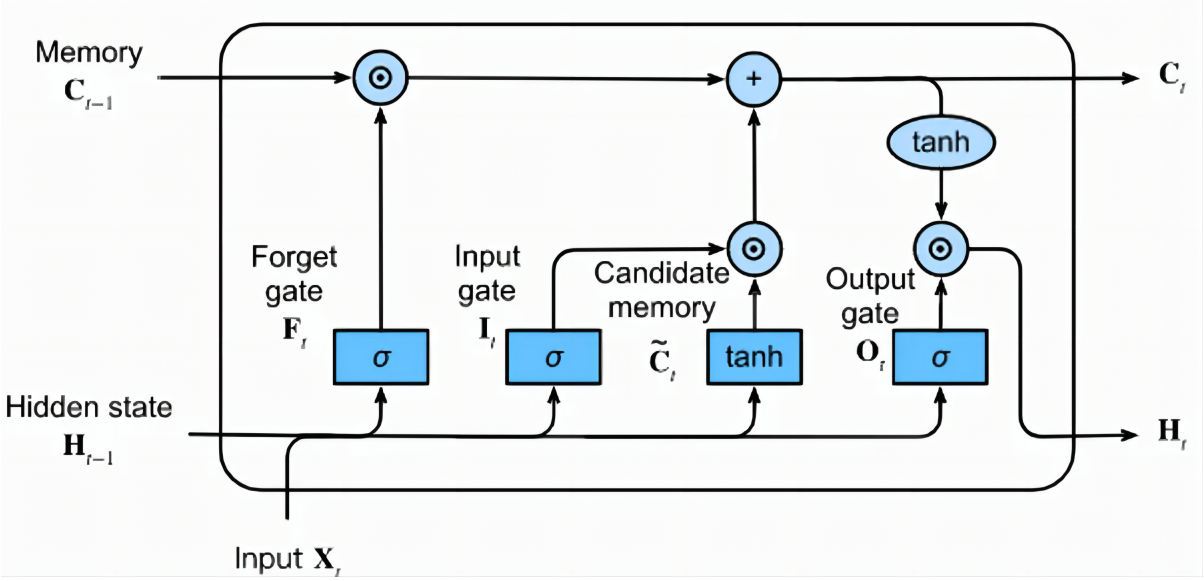


Figure 3.3.1: LSTM Structure

1. Input Gate

The input gate determines how much new information should be stored in the cell state C_t at the current time step t.

The **sigmoid layer** within the input gate decides which parts of the incoming data $[h_{t-1}, x_t]$ are important to update the cell state:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{3.31}$$

where:

- i_t: Input gate vector that controls the update of the cell state C_t.
- σ: Sigmoid activation function.
- W_i, b_i: Weight matrix and bias vector for the input gate.
- h_{t-1}: Previous hidden state.
- x_t: Current input.

The **candidate value** $\tilde{C}_t$, generated by the tanh activation function, represents the new information that could be added to C_t:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{3.32}$$

where:

- $\tilde{C}_t$: Candidate value vector.
- tanh: Hyperbolic tangent activation function.
- W_C, b_C: Weight matrix and bias vector for the candidate value computation.

The input gate thus decides both what new information to add to C_t and how much of that information to retain based on $[h_{t-1}, x_t]$.

2. Forget Gate

The forget gate controls how much of the previous cell state C_{t-1} should be retained for the current time step t.

The **sigmoid layer** within the forget gate outputs values between 0 and 1, determining which parts of C_{t-1} should be forgotten:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3.33}$$

where:

- f_t: Forget gate vector that decides what information to retain from C_{t-1}.
- σ: Sigmoid activation function.
- W_f, b_f: Weight matrix and bias vector for the forget gate.

The forget gate regulates the flow of information from C_{t-1} to C_t based on $[h_{t-1}, x_t]$.

3. Cell State Update

Once the input and forget gates have been applied, the cell state C_t is updated:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{3.34}$$

where:

- C_t: Updated cell state at time t.
- f_t: Forget gate vector.
- C_{t-1}: Previous cell state.
- i_t: Input gate vector.
- $\tilde{C}_t$: Candidate value vector.

This update mechanism combines the preserved information from C_{t-1} (controlled by f_t) and the new information (determined by i_t and $\tilde{C}_t$).

4. Output Gate

Finally, the output gate determines the actual output hidden state h_t based on the updated cell state C_t.

The **sigmoid layer** within the output gate decides which parts of C_t should be exposed as the output:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{3.35}$$

where:

- o_t: Output gate vector that controls which parts of C_t influence h_t.
- σ: Sigmoid activation function.
- W_o, b_o: Weight matrix and bias vector for the output gate.

The **output hidden state** h_t is computed by scaling the tanh activation of C_t by o_t:

$$h_t = o_t \cdot \tanh(C_t) \tag{3.36}$$

where:

- h_t: Output hidden state at time t.
- tanh: Hyperbolic tangent activation function.
- C_t: Updated cell state.

LSTM networks utilize these gated mechanisms to effectively capture and utilize long-term dependencies in sequential data. The input gate decides what new information to incorporate into the cell state, the forget gate controls what information to retain from the previous cell state, and the output gate determines the output hidden state based on the current cell state.

3.3.1.2 Applications in Time Series Prediction

Time series prediction is a critical task in various fields such as finance, weather forecasting, and stock market analysis. Long Short-Term Memory (LSTM) networks, a variant of Recurrent Neural Networks (RNNs), have shown remarkable success in this domain due to their ability to model long-term dependencies. In this section, we introduce the application of LSTM in time series prediction, detailing the steps from data preprocessing to model evaluation.

1. Data Preprocessing

Data preprocessing is a crucial step in time series prediction. Data cleaning is necessary to handle outliers and missing values. Normalization or scaling (e.g., MinMax scaling) is also applied to expedite the convergence of the model during training.[58] Min-Max scaling is a fundamental technique in data preprocessing that adjusts the scale of numerical features to a specified range, typically $[0, 1]$ or $[-1, 1]$. This method is widely used in machine learning to normalize data and improve model performance.

The Min-Max scaling formula is expressed as:

$$X_{\text{scaled}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \tag{3.37}$$

Where:

- X is the original value of the feature.
- X_{min} is the minimum value of the feature.
- X_{max} is the maximum value of the feature.

This formula ensures that X_{scaled} falls within the range defined by $[0, 1]$ or $[-1, 1]$, depending on the specific requirements.

Min-Max scaling offers several advantages:

- **Normalization**: It scales features to a specific range, facilitating easier comparison between different features.
- **Preservation of Data Distribution**: It preserves the relative relationships between data points, maintaining the original distribution of the data.
- **Improved Model Performance**: Scaling features to a consistent range can improve the convergence speed and performance of machine learning algorithms, especially those sensitive to the scale of input data.

By transforming features to a fixed range, Min-Max scaling ensures that data is on a comparable scale, which is essential for accurate and effective modeling. Applying Min-Max scaling correctly can significantly enhance the performance and reliability of machine learning models across various domains and applications.

2. Data Splitting

After pre-processing, the time series data is divided into training and testing sets to evaluate and validate the model's performance. A common approach is to use a sliding window method, where the training set encompasses earlier time steps, and the testing set includes subsequent time steps. This methodology ensures that the model can effectively generalize to unseen future data.

The sliding window approach divides the time series data into overlapping subsequences, typically for training and testing purposes. Let's denote the time series data as $\{X_t\}_{t=1}^{T}$, where T is the total number of time steps. Suppose we want to use a sliding window of size N for training and testing.

The training set $\mathcal{D}_{\text{train}}$ consists of subsequences from the beginning of the time series up to a certain point T_{train}:

$$\mathcal{D}_{\text{train}} = \{(X_{t-N+1}, X_{t-N+2}, \ldots, X_t) \mid N \leq t \leq T_{\text{train}}\} \tag{3.38}$$

Here, N is the window size, and T_{train} is the index where the training data ends.

The testing set $\mathcal{D}_{\text{test}}$ includes subsequences from $T_{\text{train}} + 1$ to the end of the time series:

$$\mathcal{D}_{\text{test}} = \{(X_{t-N+1}, X_{t-N+2}, \ldots, X_t) \mid T_{\text{train}} < t \leq T\} \tag{3.39}$$

This ensures that the testing data contains future time steps that the model has not seen during training, allowing for a realistic evaluation of its predictive capability.

The sliding window approach offers several benefits for time series analysis:

- **Temporal Order Preservation**: The sliding window method maintains the sequential order of time series data, crucial for time-dependent modeling tasks.
- **Generalization Evaluation**: By using earlier data for training and later data for testing, the model's ability to generalize to unseen future data can be accurately assessed.
- **Flexibility**: Adjusting the window size N allows for flexibility in capturing short-term or long-term dependencies in the time series.

The sliding window approach is a fundamental technique for partitioning time series data into training and testing sets. It ensures that models can effectively learn from historical data and generalize to predict future observations. Choosing an appropriate window size N is essential and depends on the specific characteristics and patterns present in the time series data.

3. Building and Training the LSTM Model

Long Short-Term Memory (LSTM) networks, a variant of Recurrent Neural Networks (RNNs), are particularly adept at handling time series data due to their ability to capture long-term dependencies. LSTMs mitigate the issues of vanishing and exploding gradients that traditional RNNs face through their unique architecture, which includes memory cells and gating mechanisms such as the forget gate, input gate, and output gate.

A detailed explanation of the LSTM model has already been provided in section 3.3.1.1.

4. Model Prediction

After training, the LSTM model can be used to make predictions on the test set. Leveraging the persistence of its memory cells and gating mechanisms, the LSTM is capable of capturing long-term dependencies in the time series data, thus providing accurate forecasts for future time steps.

In the prediction phase, the trained LSTM model processes the input sequence to forecast future values. Given a trained LSTM model and an input sequence $X = \{x_1, x_2, \ldots, x_T\}$, the model predicts the next time step y_{T+1} by leveraging the learned parameters and the final hidden state h_T.

The prediction process involves three main steps, which are input sequence processing, hidden state calculation, and output generation.[59]

Input Sequence Processing

The input sequence $X = \{x_1, x_2, \ldots, x_T\}$ is fed into the LSTM model one time step at a time. For each time step t, the model updates its hidden state h_t and cell state C_t based on the current input x_t, the previous hidden state h_{t-1}, and the previous cell state C_{t-1}.

Hidden State Calculation

The hidden state h_t at each time step t is computed through a series of steps involving the input gate, forget gate, candidate cell state, and output gate. The specific calculations are as follows:

(a) Forget Gate Calculation

The forget gate determines the extent to which the previous cell state C_{t-1} should be carried forward. It is calculated using:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3.40}$$

where W_f is the weight matrix, b_f is the bias for the forget gate, $[h_{t-1}, x_t]$ is the concatenation of the previous hidden state and the current input, and σ is the sigmoid function.

(b) Input Gate Calculation

The input gate determines the amount of new information from the current input x_t and the previous hidden state h_{t-1} that should be added to the cell state. It is calculated using:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{3.41}$$

where W_i is the weight matrix, b_i is the bias for the input gate.

(c) Candidate Cell State Calculation

The candidate cell state $\tilde{C}_t$ represents the new information that could be added to the cell state. It is computed using:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{3.42}$$

where W_C is the weight matrix, b_C is the bias for the cell state, and tanh is the hyperbolic tangent function.

(d) Cell State Update

The cell state C_t is updated by combining the previous cell state C_{t-1}, modulated by the forget gate f_t, and the candidate cell state $\tilde{C}_t$, modulated by the input gate i_t:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{3.43}$$

where $\odot$ denotes element-wise multiplication.

(e) Output Gate Calculation

The output gate determines the extent to which the updated cell state C_t should influence the hidden state h_t. It is calculated using:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{3.44}$$

where W_o is the weight matrix, b_o is the bias for the output gate.

(f) Hidden State Update

Finally, the hidden state h_t is updated using the output gate o_t and the updated cell state C_t:

$$h_t = o_t \odot \tanh(C_t) \tag{3.45}$$

These calculations ensure that the hidden state h_t at each time step t captures both the short-term input and long-term dependencies, enabling the LSTM model to maintain relevant historical information over long sequences.

Output Generation

After processing the entire input sequence, the LSTM model uses the final hidden state h_T to generate the prediction for the next time step. The output y_{T+1} is computed using a dense layer that maps the hidden state to the prediction space. This is typically achieved through a linear transformation followed by an activation function ϕ. The equation for the output generation is:

$$y_{T+1} = \phi(W_y \cdot h_T + b_y) \tag{3.46}$$

where W_y is the weight matrix, b_y is the bias term, and ϕ is the activation function applied to the output layer. In many cases, ϕ could be a linear function for regression tasks or a softmax function for classification tasks.

Overall, the detailed prediction process of the LSTM model, driven by its ability to maintain and update hidden and cell states over long sequences, ensures accurate and robust forecasts for future time steps. This capability makes LSTM models highly effective for time series forecasting tasks.

5. Model Evaluation

Evaluating the LSTM model's predictive performance is critical. Common evaluation metrics include Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE). These metrics quantify the deviation between the predicted values and the actual observed values, helping to assess the accuracy and effectiveness of the model.

Mean Squared Error is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.47}$$

Root Mean Squared Error is the square root of MSE and is given by:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{3.48}$$

Mean Absolute Error is defined as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3.49}$$

where:

- n is the number of samples,
- y_i is the actual observed value for sample i,
- $\hat{y}_i$ is the predicted value for sample i.

These metrics provide a comprehensive evaluation of the LSTM model's predictive performance, emphasizing different aspects of the prediction accuracy. MSE penalizes large errors more significantly due to squaring, while RMSE provides an interpretable measure of error in the same units as the predicted values. MAE offers a robust measure of average error magnitude, suitable for scenarios with non-Gaussian error distributions or presence of outliers.

By following these steps, LSTM networks demonstrate their robust capability in time series prediction, especially in capturing complex, long-term dependencies. The theoretical aspects highlight how LSTM's unique structure and optimization techniques effectively process time series data, enhancing prediction accuracy and generalization.

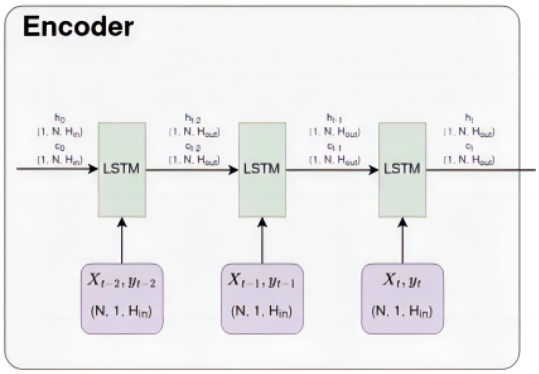
3.3.2 Single-sensor LSTM Encoder-Decoder Model

The Single-sensor LSTM Encoder-Decoder Model is a deep learning-based sequence prediction model designed specifically for handling time series data. Leveraging the capabilities of Long Short-Term Memory (LSTM) networks, it effectively captures long-term dependencies and patterns within time series.[60]

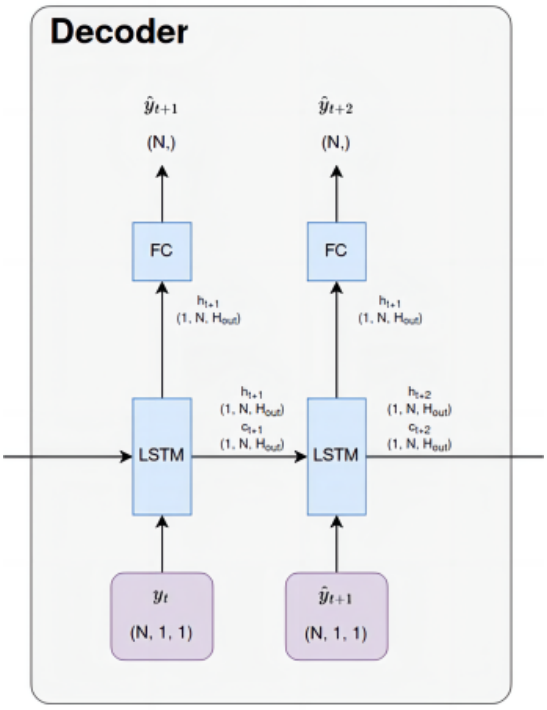
3.3.2.1 Single-sensor LSTM Encoder-Decoder Model Structure

This section outlines the key components and structure of the Single-sensor LSTM Encoder-Decoder Model, which is crucial for understanding its functionality and application in time series analysis.

The overall structure of the Single-sensor LSTM Encoder-Decoder Model is shown in figure 3.3.2. The model consists of an LSTM encoder and a decoder . The encoder consists of three LSTM layers that are used to compress the input sequence into a context vector. The decoder, also composed of LSTM layers, receives the context vector and generates the output sequence step-by-step, with the prediction at each time step relying on the previous prediction and the context vector.[61]



(a) Single-sensor LSTM Encoder-Decoder Model (Encoder)



(b) Single-sensor LSTM Encoder-Decoder Model (Decoder)

Figure 3.3.2: Single-sensor LSTM Encoder-Decoder Model

1. Encoder

In sequence-to-sequence models, the encoder plays a pivotal role in processing input sequences to generate a context vector that encapsulates the entire input information. Here, we delve into the architecture of the encoder and its integration with LSTM (Long Short-Term Memory) cells.

The encoder receives a sequence of input vectors $X_{t-2}, X_{t-1}, X_t, \ldots$, where each $X_i \in \mathbb{R}^{N\times 1\times Hin}$ represents a data point at time i, with N denoting the batch size and Hin denoting the input feature size. These vectors represent the temporal evolution of input data.

The encoder employs LSTM cells to sequentially process the input sequence. An LSTM cell comprises several components: input gate i_t, forget gate f_t, cell state c_t, output gate o_t, and hidden state h_t. The specific structure of lstm is shown in the following equations. The specific structure is described in section 3.3.1.

(a) Input Gate i_t: The input gate i_t is computed as:

$$i_t = \sigma(W_{ii}X_i + W_{hi}h_{i-1} + b_i) \tag{3.50}$$

The input gate controls how much of the current input X_i should be added to the cell state c_i. It computes a sigmoid activation σ based on the current input X_i, previous hidden state h_{i-1}, and biases b_i.

(b) Forget Gate f_t: The forget gate f_t is computed as:

$$f_t = \sigma(W_{if}X_i + W_{hf}h_{i-1} + b_f) \tag{3.51}$$

The forget gate controls how much of the previous cell state c_{i-1} should be retained. Similar to the input gate, it uses a sigmoid activation σ to decide which parts of the previous cell state should be forgotten.

(c) Input Modulation g_t: The input modulation g_t is computed as:

$$g_t = \tanh(W_{ig}X_i + W_{hg}h_{i-1} + b_g) \tag{3.52}$$

The input modulation gate determines new candidate values that could be added to the cell state c_i. It calculates a hyperbolic tangent tanh of a linear combination of X_i and h_{i-1}, adjusted by weights W_{ig}, W_{hg}, and bias b_g.

(d) Output Gate o_t: The output gate o_t is computed as:

$$o_t = \sigma(W_{io}X_i + W_{ho}h_{i-1} + b_o) \tag{3.53}$$

The output gate controls how much of the cell state c_i should be exposed as the output h_i. It uses a sigmoid activation σ to determine which parts of the cell state c_i should contribute to the output at the current time step.

(e) Cell State Update c_i: The cell state c_i is updated as:

$$c_i = f_t \odot c_{i-1} + i_t \odot g_t \tag{3.54}$$

Update the cell state c_i based on the forget gate f_t and input gate i_t. $\odot$ denotes element-wise multiplication. Combine the retained parts of c_{i-1} and the newly computed candidate values g_t.

(f) Hidden State h_i: The hidden state h_i is computed as:

$$h_i = o_t \odot \tanh(c_i) \tag{3.55}$$

The hidden state h_i is the output of the LSTM cell. Compute the hyperbolic tangent tanh of c_i and scale it by o_t (element-wise multiplication).

As the encoder processes each input vector X_i through the LSTM cells, it updates its hidden state h_i and cell state c_i. The final hidden state h_t after processing the entire sequence serves as the context vector c_t. This context vector c_t represents a distilled representation of the entire input sequence, encapsulating relevant information needed for subsequent decoding.

The encoder integrates LSTM cells to effectively capture long-term dependencies and temporal patterns present in sequential data. The input gate i_t controls how much new information is added to the cell state, the forget gate f_t regulates the extent of information to be discarded from the previous cell state, and the output gate o_t determines the form of the output. This gating mechanism allows the encoder to robustly encode and retain essential features from the input sequence.

2. Decoder

The decoder generates an output sequence based on the encoded representation of the input sequence. The decoder starts with the context vector c_t obtained from the encoder, which serves as the initial state for generating the output sequence.

The decoder consists of a stack of LSTM (Long Short-Term Memory) cells. At each time step i, an LSTM cell receives:

- **Previous Hidden State**: h_{i-1}, representing the decoder's state from the previous time step.
- **Previous Predicted Output**: $\hat{y}_{i-1}$, the decoder's output prediction at the previous time step.
- **Context Vector**: c_t, derived from the encoder and providing a global representation of the input sequence.

The LSTM cell computes a new hidden state h_i and cell state c_i based on the following equations:

(a) Forget gate

$$f_i = \sigma(W_f[\hat{y}_{i-1}, h_{i-1}, c_t] + b_f) \tag{3.56}$$

The forget gate f_i determines how much of the previous cell state c_{i-1} should be retained or forgotten. It takes as input the concatenation $[\hat{y}_{i-1}, h_{i-1}, c_t]$ of the previous output $\hat{y}_{i-1}$, previous hidden state h_{i-1}, and current context c_t. This concatenated vector is linearly transformed by weights W_f and a bias term b_f, followed by a sigmoid activation σ to output values between 0 and 1. The closer the output is to 1, the more of c_{i-1} is retained in c_i.

(b) Input gate

$$i_i = \sigma(W_i[\hat{y}_{i-1}, h_{i-1}, c_t] + b_i) \tag{3.57}$$

The input gate i_i determines how much of the candidate values $\tilde{c}_i$ should be added to the cell state c_i. It computes a sigmoid activation over a linear transformation of the concatenated vector $[\hat{y}_{i-1}, h_{i-1}, c_t]$ using weights W_i and a bias term b_i. The sigmoid activation decides which elements of $\tilde{c}_i$, computed as $\tanh(W_c[\hat{y}_{i-1}, h_{i-1}, c_t]+b_c)$, should be updated into c_i.

(c) Candidate cell state

$$\tilde{c}_i = \tanh(W_c[\hat{y}_{i-1}, h_{i-1}, c_t] + b_c) \tag{3.58}$$

The candidate cell state $\tilde{c}_i$ represents the new candidate values that could be added to the cell state c_i. It is computed by applying a hyperbolic tangent activation function tanh to a linear transformation of the concatenated vector $[\hat{y}_{i-1}, h_{i-1}, c_t]$ using weights W_c and a bias term b_c. The tanh function outputs values between -1 and 1, making it suitable for regulating the cell state.

(d) Cell state

$$c_i = f_i \odot c_{i-1} + i_i \odot \tilde{c}_i \tag{3.59}$$

The cell state c_i is updated based on the forget gate f_i, the previous cell state c_{i-1}, the input gate i_i, and the candidate cell state $\tilde{c}_i$. Element-wise operations are performed here: $\odot$ denotes element-wise multiplication. $f_i \odot c_{i-1}$ scales the previous cell state values by the forget gate, effectively deciding which parts of c_{i-1} should be retained. $i_i \odot \tilde{c}_i$ scales the candidate values by the input gate, determining how much of $\tilde{c}_i$ should be added to c_i.

(e) Output gate

$$o_i = \sigma(W_o[\hat{y}_{i-1}, h_{i-1}, c_t] + b_o) \tag{3.60}$$

The output gate o_i determines which parts of the cell state c_i should be exposed as the output h_i of the LSTM cell. It computes a sigmoid activation over a linear transformation of the concatenated vector $[\hat{y}_{i-1}, h_{i-1}, c_t]$ using weights W_o and a bias term b_o. The sigmoid activation scales the values in c_i to produce h_i.

(f) Hidden state

$$h_i = o_i \odot \tanh(c_i) \tag{3.61}$$

The hidden state h_i is the output of the LSTM cell for time step i. It is computed by element-wise multiplying the output gate o_i with the hyperbolic tangent activation tanh of the cell state c_i. The tanh function scales c_i to produce values between -1 and 1, which are then modulated by o_i to produce the final hidden state output h_i.

After each LSTM cell, there is a fully connected layer that maps the decoder's hidden state h_i to a prediction vector $\hat{y}_i$:

$$\hat{y}_i = \text{softmax}(W_s h_i + b_s) \tag{3.62}$$

where W_s is the weight matrix and b_s is the bias vector of the fully connected layer. The softmax function normalizes the output into a probability distribution over the vocabulary.

The decoder generates predictions iteratively for subsequent time steps i, using $\hat{y}_{i-1}$ as input to predict $\hat{y}_i$. This process continues until an end-of-sequence token is generated or a maximum sequence length is reached.

In summary, the decoder in a Seq2Seq model utilizes LSTM cells to decode the context vector from the encoder and generate output predictions iteratively. This architecture enables the model to effectively capture dependencies and generate sequences of varying lengths.

3.3.2.2 Applications in Time Series Prediction

In this section, the application process of LSTM Encoder-Decoder architecture for time series prediction will be introduced.

1. Data Pre-processing

The data pre-processing process is the same as in chapter 3.3.3.1 and is not described here, see chapter 3.3.1.2 for details.

2. Building and training the LSTM Encoder-Decoder Model

The LSTM Encoder-Decoder model is structured to effectively capture temporal dependencies in sequential data. It consists of an encoder to process input sequences and a decoder to generate output sequences based on the encoded information.

Encoder:

- **LSTM Layers:** Each LSTM layer computes the hidden state h_t and cell state c_t at time step t:

$$h_t, c_t = \text{LSTM}_{\text{encoder}}(x_t, h_{t-1}, c_{t-1}) \tag{3.63}$$

  where x_t is the input at time step t, h_{t-1} and c_{t-1} are the previous hidden and cell states.

Decoder:

- **LSTM Layers:** Similarly, the decoder LSTM computes its hidden and cell states using the context vector h_{encoder}:

$$h_t, c_t = \text{LSTM}_{\text{decoder}}(y_{t-1}, h_{t-1}, c_{t-1}) \tag{3.64}$$

  where y_{t-1} is the input (or the previous output during training).

The choice of loss function is crucial in training the LSTM Encoder-Decoder model for time series prediction tasks. Mean Squared Error (MSE) is commonly used to measure the difference between predicted and actual values.

To minimize the MSE and optimize the LSTM Encoder-Decoder model parameters, an appropriate optimizer is selected. Adam and RMSprop are popular choices due to their efficiency in handling large-scale optimization tasks.[62]

Adam Optimizer:

- Update rule for parameters θ:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.65}$$

  where η is the learning rate, $\hat{m}_t$ and $\hat{v}_t$ are estimates of the first moment (mean) and the second moment (uncentered variance) of the gradients, and ϵ is a small constant to prevent division by zero.

RMSprop Optimizer:

- Update rule:

$$v_t = \gamma v_{t-1} + (1 - \gamma)(\nabla_\theta L(\theta_t))^2 \tag{3.66}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot \nabla_\theta L(\theta_t) \tag{3.67}$$

  where γ is a decay factor, η is the learning rate, ϵ is a small constant to prevent division by zero, and $L(\theta_t)$ is the loss function at iteration t.

The training process involves iteratively adjusting the model parameters to minimize the chosen loss function. This process includes forward and backward passes through the LSTM Encoder-Decoder model, computing gradients, and updating weights using the optimizer.

- **Forward Pass:** Compute predictions $\hat{y}_i$ using the encoder-decoder architecture.
- **Calculate Loss:** Compute MSE loss between predictions $\hat{y}_i$ and actual values y_i.
- **Backward Pass (Backpropagation):** Compute gradients of the loss function with respect to the model parameters using the chain rule.
- **Optimization Step:** Update model parameters using the chosen optimizer.
- **Repeat:** Iterate through the training data multiple times (epochs) to improve model performance.

3. Prediction Process

During time series prediction, an encoder-decoder model is used to forecast future values based on historical data. This involves encoding the historical information and decoding it to generate future predictions.[63]

(a) Encoder Phase:

- **Encoding Historical Data:** The encoder processes the historical sequence $X_{1:T} = (x_1, x_2, \ldots, x_T)$, where x_t represents the data at time step t, into a fixed-dimensional representation h_T:

$$h_T = \text{Encoder}(X_{1:T}) \tag{3.68}$$

  Here, h_T encapsulates the relevant information extracted from $X_{1:T}$ and serves as the context or latent representation for prediction. The encoder function Encoder typically involves recurrent neural networks (RNNs), such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU), or transformer architectures, which effectively capture temporal dependencies and patterns in the time series data.

- **Recurrent Neural Network (RNN):** For RNN-based encoders, the latent representation h_T is computed recursively over time steps t:

$$h_t = \text{RNN}(x_t, h_{t-1}) \tag{3.69}$$
$$h_T = h_T \quad \text{(after processing all time steps)} \tag{3.70}$$

  Here, RNN denotes the recurrent neural network function, x_t is the input at time step t, and h_t is the hidden state at time step t.

- **Transformer Architecture:** Alternatively, transformer-based encoders use self-attention mechanisms to capture dependencies across the entire sequence $X_{1:T}$:

$$h_T = \text{TransformerEncoder}(X_{1:T}) \tag{3.71}$$

The TransformerEncoder processes the input sequence $X_{1:T}$ by attending to all positions in the sequence simultaneously, allowing for parallel computation of representations.

(b) Decoder Phase:

- **Generating Future Predictions:** The decoder utilizes h_T, the encoded representation of historical data $X_{1:T}$, to generate predictions $\hat{Y}_{T+1:T+k}$ for future time steps $T+1$ to $T+k$:

$$\hat{Y}_{T+1:T+k} = [\hat{y}_{T+1}, \hat{y}_{T+2}, \ldots, \hat{y}_{T+k}] = \text{Decoder}(h_T) \tag{3.72}$$

  Here, $\hat{y}_{T+t}$ represents the predicted value at time step $T+t$. The decoder function Decoder initializes its internal state using h_T and sequentially generates predictions for each future time step within the forecasting horizon k.

- **Prediction Process:** At each time step t, the output $\hat{y}_{T+t}$ is computed based on the previous output $\hat{y}_{T+t-1}$ and possibly previous ground truth values (during training), using the decoder function Decoder:

$$\hat{y}_{T+t} = \text{Decoder}(\hat{y}_{T+t-1}, h_T) \tag{3.73}$$

  This recursive process starts with an initial input $\hat{y}_{T+1}$, which can be set to zero or a special start symbol during inference, and continues iteratively to generate subsequent predictions. Each prediction $\hat{y}_{T+t}$ influences the computation of the next prediction, ensuring that the model's predictions are coherent and consistent over time.

- **Training with Ground Truth:** During training, the decoder may use ground truth values $y_{T+1:T+k}$ from the training data to adjust its internal states and parameters θ_{dec} accordingly, optimizing the model to minimize prediction errors:

$$\mathcal{L} = \sum_{t=1}^{k} \text{Loss}(\hat{y}_{T+t}, y_{T+t}) \tag{3.74}$$

  Here, $\mathcal{L}$ denotes the loss function that measures the discrepancy between predicted values $\hat{y}_{T+t}$ and actual ground truth values y_{T+t}.

The application process of LSTM Encoder-Decoder in time series forecasting involves encoding historical data into a context vector and then using a decoder to generate predictions step-by-step. This approach effectively captures long-term dependencies in time series data and adapts to various forecasting tasks.

4. Model Evaluation

The model validation method is the same as the model validation method in chapter 3.3.1.2 and will not be described here, see chapter 3.3.1.2 for details.

3.4 Gaussian Process Regression Based Sensor Fusion

Gaussian Process Regression (GPR) is a non-parametric Bayesian approach used for regression tasks. It models data by assuming that the data points come from a multivariate normal distribution. A sensor fusion model based on GPR utilizes the outputs from multiple sensors, integrating them through a Gaussian process to enhance measurement accuracy and robustness.

3.4.1 Gaussian Process Regression (GPR) Model

To understand the sensor fusion model, it is crucial to first grasp the fundamentals of Gaussian Process Regression.[64]

A Gaussian process (GP) is defined by its mean function $\mu(\mathbf{x})$ and covariance function $k(\mathbf{x}, \mathbf{x}')$:

$$f(\mathbf{x}) \sim \mathcal{GP}(\mu(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{3.75}$$

where:

- $\mu(\mathbf{x})$ is the mean function of the input $\mathbf{x}$:

$$\mu(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] \tag{3.76}$$

- $k(\mathbf{x}, \mathbf{x}')$ is the covariance function between inputs $\mathbf{x}$ and $\mathbf{x}'$:

$$k(\mathbf{x}, \mathbf{x}') = \text{Cov}(f(\mathbf{x}), f(\mathbf{x}')) \tag{3.77}$$

Given these definitions, a Gaussian process can be used to model distributions over functions. For any finite set of input points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, the corresponding function values $\mathbf{f} = [f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_n)]$ follow a multivariate normal distribution:

$$\mathbf{f} \sim \mathcal{N}(\mu, \mathbf{K}) \tag{3.78}$$

where μ is the mean vector and $\mathbf{K}$ is the covariance matrix given by:

$$\mu = \begin{bmatrix} \mu(\mathbf{x}_1) \\ \mu(\mathbf{x}_2) \\ \vdots \\ \mu(\mathbf{x}_n) \end{bmatrix} \tag{3.79}$$

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \cdots & k(\mathbf{x}_1, \mathbf{x}_n) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \cdots & k(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & k(\mathbf{x}_n, \mathbf{x}_2) & \cdots & k(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix} \tag{3.80}$$

In addition, when selecting the covariance function for a Gaussian process, it directly influences the behavior and predictive capabilities of the model.[65] Here are some common covariance functions with their specific forms:

1. Squared Exponential Kernel (RBF Kernel):

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2l^2}\right) \tag{3.81}$$

Here, σ_f^2 is the variance parameter, and l is the length scale parameter. This kernel measures the similarity between input vectors $\mathbf{x}$ and $\mathbf{x}'$ based on the Euclidean distance. Smaller distances result in larger covariance, indicating stronger correlation between their outputs. The RBF kernel is commonly used for modeling smooth and continuous functions.

2. Linear Kernel:

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}' \tag{3.82}$$

This kernel computes the dot product between input vectors $\mathbf{x}$ and $\mathbf{x}'$. It assumes that the relationship between outputs at different inputs is linear, suitable for capturing linear relationships among input variables.

In addition to these kernels, there are other types such as polynomial kernels, periodic kernels, etc., each with their own advantages and characteristics suited for different application scenarios. Choosing the appropriate covariance function allows adjustment of the Gaussian process model's predictive and generalization capabilities according to the problem's characteristics.

3.4.2 Sensor Fusion Model

With an understanding of Gaussian Process Regression, we can now explore its application in sensor fusion. Sensor fusion involves integrating data from multiple sensors to obtain more accurate and reliable information. By combining data from multiple sensors, the limitations of individual sensors can be mitigated, enhancing overall measurement accuracy and robustness.

The process of sensor fusion using GPR involves the following several key steps.[66]

1. Data Collection

The first step involves gathering measurement data from multiple sensors. Suppose there are n sensors, each providing an output $\mathbf{y}_i$, where $i = 1, 2, \ldots, n$.

During the data collection phase:

- **Sensor Deployment**: Sensors are placed strategically to capture relevant measurements in the field or designated area.
- **Data Recording**: Sensors autonomously or periodically record data, which is transmitted to a central storage or data logging system.
- **Data Integration**: Collected data from all sensors are aggregated and stored centrally for subsequent analysis and processing.

By systematically collecting data from multiple sensors, we establish the foundation for further data analysis processes, including pre-processing, feature extraction, and modeling.

2. Building Gaussian Process Models and Parameters Selection

Next, establishing a Gaussian process model for the output data of each sensor is needed.

For the output $\mathbf{y}_i$ of sensor i, we assume a Gaussian process model:

$$\mathbf{y}_i \sim \mathcal{GP}(\mu_i(\mathbf{x}), k_i(\mathbf{x}, \mathbf{x}'))$$

where:

- $\mu_i(\mathbf{x})$ is the mean function of sensor i's Gaussian process, representing the expected value of $\mathbf{y}_i$ given inputs $\mathbf{x}$.
- $k_i(\mathbf{x}, \mathbf{x}')$ is the covariance (kernel) function between input vectors $\mathbf{x}$ and $\mathbf{x}'$ for sensor i, capturing the similarity or dependence between output values at different inputs.

In the following subsections, we explore the detailed steps involved in applying Gaussian process modeling to sensor data.

(a) Selection of Mean Function $\mu_i(\mathbf{x})$

- **Choosing the Mean Function**: Select an appropriate mean function $\mu_i(\mathbf{x})$ based on prior knowledge or assumptions about the sensor's behavior. Common choices include constant functions, linear functions, or the global average of the data.

(b) Selection and Parameter Estimation of Covariance Function $k_i(\mathbf{x}, \mathbf{x}')$

- **Choosing the Covariance Function**: Select a suitable covariance function $k_i(\mathbf{x}, \mathbf{x}')$ that aligns with the characteristics of the data, such as smoothness, periodicity, or spatial correlation. Popular choices include the Gaussian (RBF) kernel, Matérn kernel, etc.
- **Parameter Estimation**: Estimate the parameters (hyperparameters) of the chosen covariance function $k_i(\mathbf{x}, \mathbf{x}')$ using techniques like maximum likelihood estimation or cross-validation. These parameters control the smoothness and scale of variations in the output predictions.

(c) Fitting the Gaussian Process

- **Model Fitting**: Fit the Gaussian process model using the selected mean and covariance functions to the observed data $\mathbf{y}_i$. This involves computing the posterior distribution of the GP, which provides predictions and uncertainties for $\mathbf{y}_i$ at new input points $\mathbf{x}^*$.

3. Data Fusion

In Gaussian Process Regression (GPR), sensor data fusion plays a crucial role as it leverages multi-source data to enhance model performance. Here, we will detail the process of sensor data fusion using a network of gas sensors located at different places.

(1) Feature Extraction and Selection

(a) Feature Extraction: Feature extraction is the process of obtaining useful information from sensor data. For gas sensors, both the location of the sensors and the gas concentration measurements are crucial features. Key types of features include:

- **Direct Features**: Using raw sensor data directly. For instance, the measurement z_i (gas concentration) and the corresponding location (x_i, y_i) for each sensor.
- **Derived Features**: Features obtained through calculations on the raw data, such as moving averages or rate of change. These features help capture trends and patterns in the data.

(b) Feature Selection: The goal of feature selection is to choose the most useful features for the model prediction. For gas sensor networks, position features (x_i, y_i) and measurement features z_i are essential. Methods like Recursive Feature Elimination (RFE) and L1 regularization can be employed to select the most relevant derived features.

(2) Feature Fusion

Feature fusion combines data from different sensors into a comprehensive feature vector. Common methods include:

- **Direct Concatenation**: Directly concatenating the feature vectors from different sensors. For example, if we have three gas sensors with outputs z_{sensor1}, z_{sensor2}, and z_{sensor3}, the combined feature vector is:

$$x = [z_{\text{sensor1}}, z_{\text{sensor2}}, z_{\text{sensor3}}] \tag{3.83}$$

  This method is simple and suitable when there are no significant differences between the sensor data.

- **Weighted Fusion**: Weighted fusion assigns different weights to the data from different sensors based on their uncertainty or confidence. This can be expressed as:

$$x = w_{\text{sensor1}} \cdot z_{\text{sensor1}} + w_{\text{sensor2}} \cdot z_{\text{sensor2}} + w_{\text{sensor3}} \cdot z_{\text{sensor3}} \tag{3.84}$$

  where w_{sensor1}, w_{sensor2}, and w_{sensor3} are the respective weights.

- **Bayesian Fusion**: Bayesian fusion uses Bayesian methods to combine prior knowledge and observational data for sensor data fusion. Assuming the sensor data are independently distributed, the joint probability distribution is the

product of the individual sensor data distributions. This method considers sensor data uncertainty and prior knowledge for more reasonable fusion.

(3) Constructing the Joint Covariance Matrix

Building the joint covariance matrix is a key step in GPR for sensor data fusion. The joint covariance matrix captures the correlations and dependencies between different sensor data. The steps are as follows:

(a) Calculating Individual Sensor Covariance Matrices: First, calculate the covariance matrix for each sensor's data. Assume we have three gas sensors located at (x_i, y_i) with measurements z_i. The covariance matrix elements are computed using a kernel function $k((x_i, y_i), (x_j, y_j))$:

$$K_{ij} = k((x_i, y_i), (x_j, y_j)) \tag{3.85}$$

Common kernel functions include:

- **RBF Kernel (Radial Basis Function Kernel)**:

$$k((x_i, y_i), (x_j, y_j)) = \sigma^2 \exp\left(-\frac{\|(x_i, y_i) - (x_j, y_j)\|^2}{2l^2}\right) \tag{3.86}$$

  where σ^2 is the signal variance, and l is the length scale parameter.

- **Matérn Kernel**:

$$k((x_i, y_i), (x_j, y_j)) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}\|(x_i, y_i) - (x_j, y_j)\|}{l}\right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}\|(x_i, y_i) - (x_j, y_j)\|}{l}\right) \tag{3.87}$$

  where ν and l are parameters, and K_ν is the modified Bessel function.

(b) Calculating Cross-Sensor Covariance: Next, calculate the covariance matrix between different sensors. This can be estimated using kernel functions or statistical methods. For three sensor datasets, the cross-sensor covariance matrix is:

$$K_{\text{cross}} = \begin{bmatrix} k_{\text{sensor1,sensor2}} & k_{\text{sensor1,sensor3}} \\ k_{\text{sensor2,sensor1}} & k_{\text{sensor2,sensor3}} \\ k_{\text{sensor3,sensor1}} & k_{\text{sensor3,sensor2}} \end{bmatrix} \tag{3.88}$$

(c) Constructing the Joint Covariance Matrix: Finally, combine the individual sensor covariance matrices and cross-sensor covariance matrices to form the joint covariance matrix. If we have three sensors with covariance matrices K_{sensor1}, K_{sensor2}, and K_{sensor3}, the joint covariance matrix is:

$$K = \begin{bmatrix} K_{\text{sensor1}} & K_{\text{cross,12}} & K_{\text{cross,13}} \\ K_{\text{cross,21}} & K_{\text{sensor2}} & K_{\text{cross,23}} \\ K_{\text{cross,31}} & K_{\text{cross,32}} & K_{\text{sensor3}} \end{bmatrix} \tag{3.89}$$

This joint covariance matrix allows us to consider the correlations and dependencies between different sensor data, enabling more accurate and reliable predictions.

By following this approach, the GPR sensor fusion model can effectively utilize multi-source sensor data, providing more accurate and reliable predictions of gas concentration levels.

4. Prediction and Fusion

In Gaussian process regression, predicting outcomes y at new input points $\mathbf{x}$ involves deriving both a predicted mean $\mu_f(\mathbf{x})$ and uncertainty (variance) $\sigma_f^2(\mathbf{x})$ of the predictions. These quantities encapsulate the model's estimation and its confidence level based on observed data.

The predicted mean $\mu_f(\mathbf{x})$ is computed using the GP framework as:

$$\mu_f(\mathbf{x}) = \mathbf{k}_*^\top (\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{y} \tag{3.90}$$

where:

- $\mathbf{k}_*$ is the vector of covariances between the new input $\mathbf{x}$ and the training inputs,
- $\mathbf{K}$ is the covariance matrix of the training inputs,
- σ_n^2 is the noise variance,
- $\mathbf{y}$ is the vector of observed training targets.

The predictive variance $\sigma_f^2(\mathbf{x})$ quantifies uncertainty in the prediction and is given by:

$$\sigma_f^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{k}_*^\top (\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1} \mathbf{k}_* \tag{3.91}$$

where $k(\mathbf{x}, \mathbf{x})$ represents the self-covariance (kernel function evaluated at $\mathbf{x}$).

These equations demonstrate how Gaussian process regression leverages prior information from training data (via $\mathbf{K}$ and $\mathbf{y}$) and accounts for uncertainty (via σ_n^2) to provide robust predictions $\mu_f(\mathbf{x})$ and $\sigma_f^2(\mathbf{x})$ at any new input $\mathbf{x}$.

The sensor fusion model based on Gaussian Process Regression offers several significant advantages:

- **Non-Parametric Model**: No need to assume a specific form for the data distribution, making it suitable for complex nonlinear systems.
- **Uncertainty Quantification**: Provides confidence intervals for predictions, allowing for the evaluation of the reliability of the results.
- **High Flexibility**: Can handle different types of sensor data and adapt to various data characteristics by choosing appropriate kernel functions.

In summary, a sensor fusion model based on Gaussian Process Regression can significantly improve the robustness and accuracy of measurement systems, especially in scenarios with high complexity and uncertainty.

Chapter 4

Work

In this section, I will describe in detail the work I did in the project, including the details and process. The work is mainly divided into four parts, namely outlier detection, sensor calibration, short-term prediction and sensor fusion.

4.1 Outlier Detection

4.1.1 ARIMA Outlier Detection

In this section, I focused on detecting outliers in CO_2 level time series data using ARIMA (AutoRegressive Integrated Moving Average) modeling and comparing it with a custom outlier detection method. The main steps of the project involved data preprocessing, ARIMA model fitting, outlier detection using residuals from the ARIMA model, and a comparative analysis of the outlier detection performance through ROC curves. The flowchart is shown in the figure 4.1.1.

I processed CO_2 level data from multiple CSV files, filtering for specific date ranges and extracting relevant time series values. The data preprocessing involved:

1. **Loading Data**: Five CSV files containing CO_2 level data were imported into the analysis environment. Each file was structured with timestamps and corresponding CO_2 measurements.
2. **Filtering Data**: The dataset was filtered to focus on a specific date (2023-12-01) and time range (07:00 to 23:00). This involved extracting data points that fell within the specified datetime range.
3. **Cleaning Data**:
    - CO_2 values were converted to a consistent numerical format suitable for analysis.
    - Missing or erroneous data points were identified and handled through imputation or removal based on data quality assessment criteria.

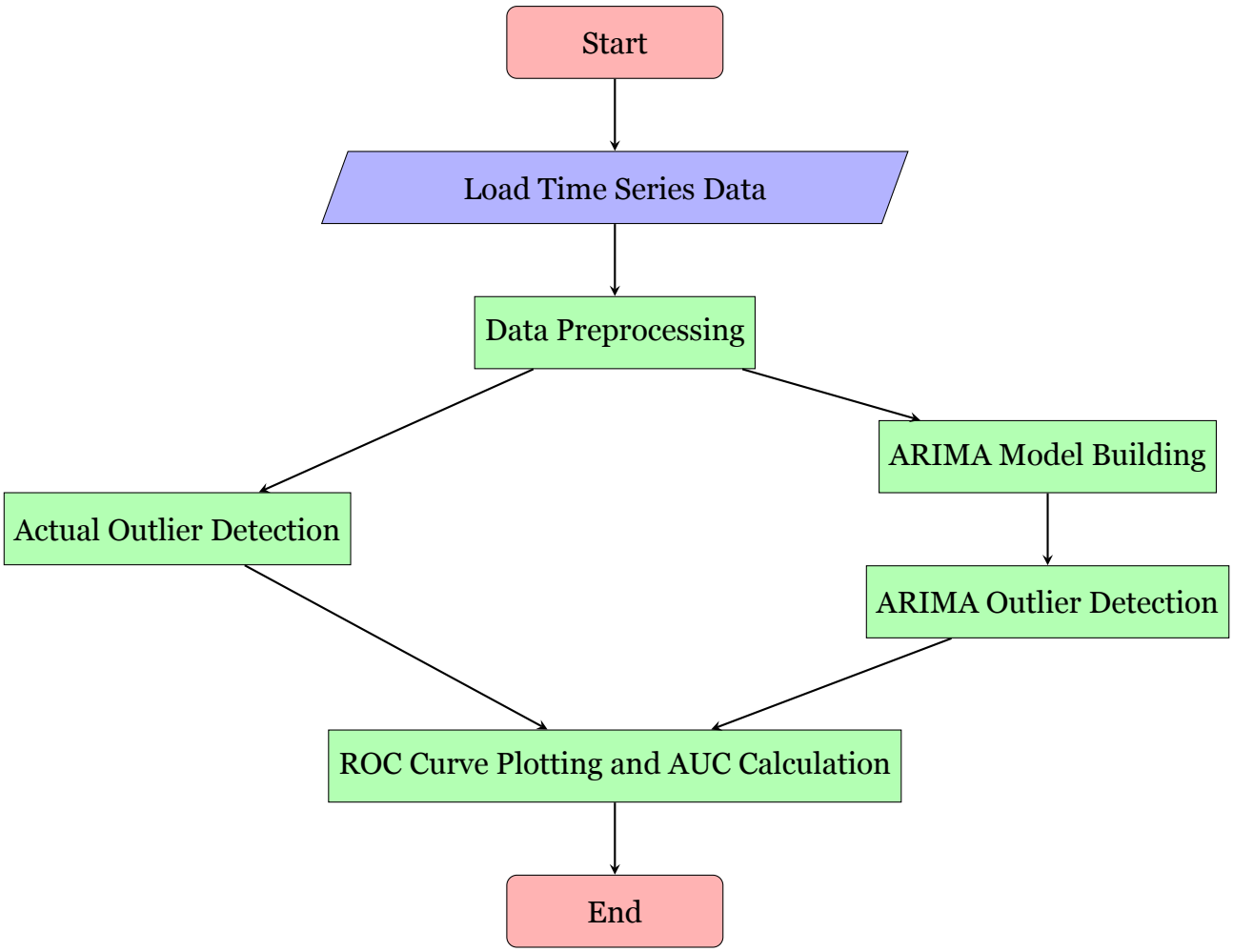


Figure 4.1.1: Flowchart of ARIMA Outlier Detection

To determine the appropriate parameters (p,d,q) for the ARIMA model, I utilized autocorrelation function (ACF) and partial autocorrelation function (PACF) plots:

1. **ACF Plot**: The ACF plot helps identify the order of the Moving Average (MA) component by plotting the correlation coefficients between the series and its lagged values. The significant lags indicate the number of MA terms.
2. **PACF Plot**: The PACF plot helps identify the order of the Autoregressive (AR) component by showing the correlations between the series and its lagged values, while controlling for the intermediate lags. Significant lags indicate the number of AR terms.

Based on these plots, I selected an ARIMA(1,1,1) model, where:

$$\nabla y_t = \phi_1 \nabla y_{t-1} + \theta_1 \epsilon_{t-1} + \epsilon_t \tag{4.1}$$

Here,

- $\nabla y_t = y_t - y_{t-1}$ is the differenced time series,

- ϕ_1 is the autoregressive coefficient,
- θ_1 is the moving average coefficient,
- ϵ_t is the error term.

The ARIMA(1,1,1) model is appropriate because it captures the first-order differencing ∇y_{t-1}, an autoregressive term ϕ_1, and a moving average term θ_1 to account for the autocorrelation and moving average patterns observed in the ACF and PACF plots.

Then, I fitted an ARIMA model to the cleaned CO_2 data:

1. **Model Selection**: I chose an ARIMA(1,1,1) model based on prior analysis. This means:
$$\nabla y_t = \phi_1 \nabla y_{t-1} + \theta_1 \epsilon_{t-1} + \epsilon_t \tag{4.2}$$
where y_t is the original time series, $\nabla y_t = y_t - y_{t-1}$ is the differenced time series, ϕ_1 is the autoregressive coefficient, θ_1 is the moving average coefficient, and ϵ_t is the error term. The model can be expanded as:
$$y_t = c + y_{t-1} + \phi_1(y_{t-1} - y_{t-2}) + \theta_1 \epsilon_{t-1} + \epsilon_t \tag{4.3}$$

2. **Fitting the Model**: Used the `statsmodels` library to fit the ARIMA model to the CO_2 time series data. The parameters ϕ_1, θ_1, and c are estimated using maximum likelihood estimation (MLE):
$$\hat{\phi}_1, \hat{\theta}_1, \hat{c} = \arg\max_{\phi_1, \theta_1, c} \sum_{t=1}^{T} \log f(y_t \mid y_{t-1}, \phi_1, \theta_1, c) \tag{4.4}$$
where $f(y_t \mid y_{t-1}, \phi_1, \theta_1, c)$ is the conditional probability density function of y_t.

3. **Predicting**: Generated in-sample predicts to compute residuals for outlier detection. The predicted value $\hat{y}_t$ is given by:
$$\hat{y}_t = c + y_{t-1} + \hat{\phi}_1(\nabla y_{t-1}) + \hat{\theta}_1 \epsilon_{t-1} \tag{4.5}$$

Example ARIMA Prediction:

Time	Actual Value	Predicted Value	Residual
1	410	411	-1
2	415	413	2
...	...	...	...
1000	400	405	-5

Table 4.1.1: Example ARIMA Prediction

I identified outliers using the residuals from the ARIMA model:

1. **Calculating Residuals**: Subtracted the ARIMA Predictions from the actual CO_2 values:
$$\text{Residual}_t = \text{Actual}_t - \text{Predict}_t \tag{4.6}$$

2. **Z-score Analysis**: Calculated the Z-scores of the residuals to identify significant deviations:
$$Z_t = \frac{\text{Residual}_t - \mu_{\text{residuals}}}{\sigma_{\text{residuals}}} \tag{4.7}$$

3. **Threshold Setting**: Defined a threshold (1.5 standard deviations) to classify points as outliers.

In addition to the ARIMA-based method, I implemented a custom outlier detection approach based on rolling mean and deviation criteria:

1. **Rolling Mean Calculation**: Computed a rolling mean with a window size of 51:
$$\text{Rolling Mean}_t = \frac{1}{51} \sum_{i=t-25}^{t+25} \text{Value}_i \tag{4.8}$$

2. **Deviation Criteria**: Marked points as outliers if they deviated significantly from the rolling mean and their immediate neighbors.

   Outlier detection criteria:

   - $|\text{Value}_t - \text{Rolling Mean}_t| > 80$
   - $|\text{Value}_t - \text{Value}_{t+1}| > 150$ or $|\text{Value}_t - \text{Value}_{t-1}| > 150$

Example outliers identified:

Index	CO_2 Value	Rolling Mean	Deviation	Outlier
30	500	400	100	Yes
150	200	310	110	Yes
...	...	...	...	...
900	450	400	50	No

Table 4.1.2: Example Outliers Identified

To evaluate the performance of the ARIMA-based outlier detection, I compared it with the custom method using ROC curves:

1. **True Outlier Labels**: Generated binary labels for true outliers based on the custom method.
2. **ARIMA Outlier Labels**: Generated binary labels for ARIMA-detected outliers based on the Z-score threshold.
3. **ROC Curve Calculation**: Used the `roc_curve` and `auc` functions to calculate the ROC curves and AUC scores for both methods.
4. **Plotting ROC Curves**: Created a plot to visualize the ROC curves and compare the performance.

The comparison of ROC (Receiver Operating Characteristic) curves highlighted the effectiveness of the ARIMA (AutoRegressive Integrated Moving Average) model in

detecting outliers in time series data. The Area Under the Curve (AUC) score quantitatively reflected the ARIMA model's performance relative to a custom outlier detection method.

Algorithm 1 delineated the detailed steps of the ARIMA outlier detection algorithm. The process began with converting the raw time series data into a suitable format for analysis, followed by the application of the ARIMA model. The fitted model was then used to predict the values of the time series, and the residuals (differences between actual and predicted values) were calculated. Z-scores of these residuals were computed to identify potential outliers. A threshold was set to distinguish significant outliers, and the detected outliers were marked.

To handle these outliers, linear interpolation was performed, which replaced the outliers with interpolated values.[67] Linear interpolation estimates the missing or outlier by assuming that the data follows a straight line between two known neighboring points. In this case, the outlier is replaced by a value that is calculated based on the trend formed by the values of the time series immediately before and after the outlier.[68] This ensures that the time series remains continuous and the integrity of the data is preserved without abrupt jumps or irregularities.

Algorithm 1 ARIMA Outlier Detection and Interpolation

Require: $data$: Time series data
Ensure: Cleaned time series data
1: Extract CO_2 values from $data$.
2: Calculate the first-order differences of the CO_2 values.
3: Plot the ACF and PACF of the differenced data.
4: Determine the ARIMA model order (p, d, q) based on the ACF and PACF plots.
5: Fit an ARIMA(p, d, q) model to the original CO_2 data.
6: Predict CO_2 values using the fitted ARIMA model.
7: Calculate residuals by subtracting predicted values from original values.
8: Calculate Z-scores of the residuals to identify outliers.
9: Replace outliers with linearly interpolated values.
10: **return** the cleaned time series data.

In this section, the application of ARIMA modeling for outlier detection in CO_2 level time series data was demonstrated. By comparing the ARIMA model with a custom outlier detection method using ROC curves, insights into the strengths and limitations of each approach were provided. The ARIMA model proved to be a robust tool for identifying significant anomalies, showcasing its potential for broader applications in time series analysis.

4.1.2 DBSACN Outlier Detection

This section explores how the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm can be employed for outlier detection in time series data.[69] DBSCAN is a density-based clustering algorithm that identifies clusters based on the density of data points within a specified radius (ϵ) and treats points that cannot be clustered as outliers (noise).[70]This article provides a comprehensive overview of DBSCAN's principles, implementation steps, and its application to outlier detection tasks in time series data. The flowchart is shown in the figure 4.1.2.

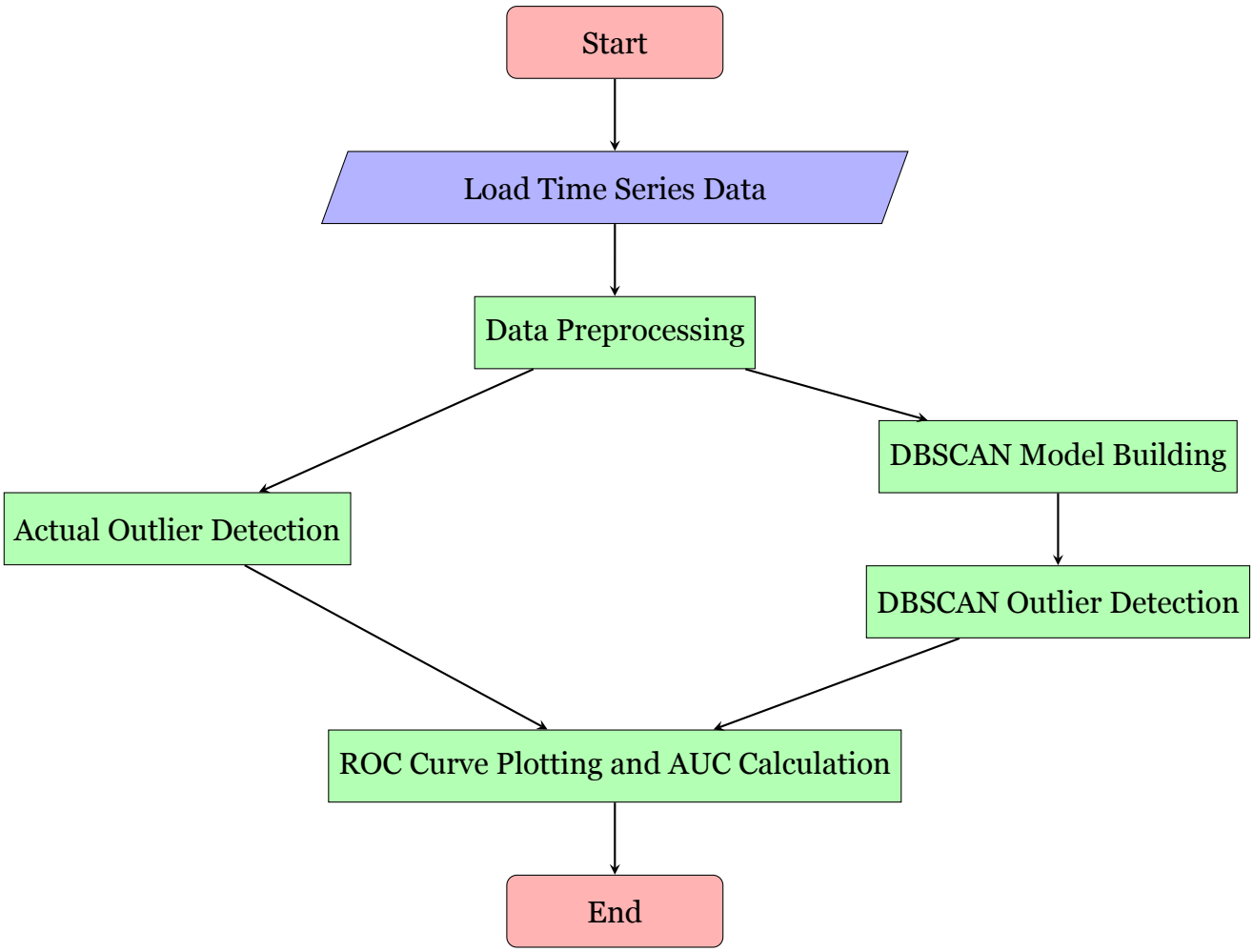


Figure 4.1.2: Flowchart of DBSCAN Outlier Detection

Before applying DBSCAN, it's essential to prepare and pre-process the data. This involves loading the data, parsing timestamps, ensuring correct data formats, and segmenting the data into fixed-length samples. For instance, data is loaded from multiple CSV files, and a specific time range is extracted for analysis. Additionally, to ensure the effectiveness of DBSCAN, the data is normalized to have zero mean and unit variance, enabling consistent distance calculations across all dimensions.

The DBSCAN algorithm is implemented in the following steps:

1. **Data Normalization**: Data is transformed to have zero mean and unit variance, which is crucial for the performance of DBSCAN:

$$\text{normalized\_data} = \frac{\text{data} - \text{data.mean()}}{\text{data.std()}} \tag{4.9}$$

   This normalization ensures that each dimension of the data is comparably weighted in distance calculations, avoiding dominance by certain dimensions.

2. **Neighborhood Definition and Core Point Labeling**:
   - **Core points**: Points that have at least `min_samples` points within a radius

ϵ are labeled as core points.

$$\text{core\_points}(p) = \{q \in D \mid \text{dist}(p, q) \leq \epsilon\} \tag{4.10}$$

Here, p denotes a data point, ϵ is the neighborhood radius, D represents the dataset.

- **Border points**: Points that have fewer than `min_samples` points within ϵ, but fall within the neighborhood of a core point.
- **Noise points**: Points that are neither core nor border points are considered noise points, i.e., outliers.

3. **Detection of Outliers**: Outliers are identified using the DBSCAN algorithm's labels (where label -1 indicates an outlier). These points are not clustered into any core or border points and are hence classified as outliers.

After identifying outliers, it's essential to handle them to maintain the integrity of the time series data. In this article, linear interpolation is used to fill in the outliers detected by DBSCAN. This approach leverages surrounding data points to interpolate values, ensuring visual continuity and preserving the original trends and variations in the time series.

To evaluate the performance of DBSCAN in outlier detection tasks, ROC curves and AUC (Area Under the Curve) are utilized to quantify its effectiveness. The ROC curve illustrates the trade-off between true positive rate (TPR, sensitivity) and false positive rate (FPR):

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{4.11}$$

Here, TPR represents the true positive rate (sensitivity), FPR is the false positive rate, TP denotes true positives, FN is false negatives, FP signifies false positives, and TN stands for true negatives. A higher AUC value indicates better performance of the algorithm.

This section provides a detailed exploration of applying the DBSCAN algorithm for outlier detection in time series data. From data preprocessing to the implementation steps of DBSCAN, followed by outlier handling and performance evaluation, each step aims to assist readers in understanding how modern data analysis techniques can be employed to handle and analyze complex time series data.

4.2 Sensor Calibration

4.2.1 Supervised Sensor Calibration

Supervised sensor calibration involves using machine learning models to enhance the precision of sensor data by training on high-precision data and low-precision data from different sensors. This section focuses on the application of the deep calibration method (DeepCM) model for supervised sensor calibration, detailing data preprocessing, model architecture, training process, and evaluation metrics. The flowchart is shown in the figure 4.2.1.

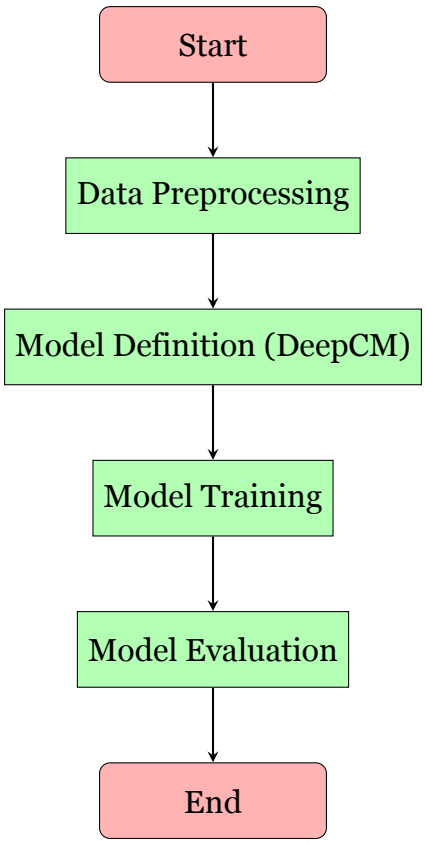


Figure 4.2.1: Flowchart of DeepCM Sensor Calibration

The sensor calibration pseudo-code for the DeepCM model is shown in Algorithm 2.

Sensor data preprocessing and preparation involve reading data from CSV files, selecting specific time ranges, removing outliers, and applying moving averages to smooth the data.

1. **Data Preprocessing**:
    - Specific time ranges are selected based on date and time
    - Outliers are removed and moving averages are applied to smooth the data, reducing noise.
    - Data is categorized into two types: Sunrise (`sunrise_vals`) and LP8 (`lp8_vals`).
2. **Data Normalization**:
    - Sunrise and LP8 data are normalized using min-max normalization to ensure data is within a consistent numerical range.

Algorithm 2 DeepCM Calibration

Require: Low-precision data (X_l), High-precision data (X_h), window size (w), skip period (p)
Ensure: Calibrated data ($\hat{X}_h$)
Data Preprocessing
Normalize X_l and X_h
Create windowed dataset: ($X_{l,w}$, $X_{h,w}$)
Model Training
Initialize DeepCM model with encoder and decoder
Define loss function (MSE) and optimizer (Adam)
for each epoch **do**
    for each batch ($x_{l,w}$, $x_{h,w}$) in training data **do**
        $h_{enc} \leftarrow$ Encoder($x_{l,w}$)
        $\hat{x}_{h,w} \leftarrow$ Decoder(h_{enc}, $x_{l,w}$[-1])
        Calculate loss: L = MSE($\hat{x}_{h,w}$, $x_{h,w}$)
        Update model parameters using L
    end for
end for
Model Evaluation
Load trained DeepCM model
Normalize new test data: $X_{l,test}$
Create windowed test data: ($X_{l,w,test}$)
$\hat{X}_{h,test} \leftarrow$ Model($X_{l,w,test}$)
Denormalize $\hat{X}_{h,test}$
Visualize and calculate RMSE of $\hat{X}_{h,test}$ against $X_{h,test}$

The DeepCM model integrates convolutional layers, gated recurrent units (GRUs), and a recurrent skip connection to effectively process time series data. This section provides an integrated explanation of the model components.[71]

1. **Convolutional Layer**:
   The convolutional layer applies filters to extract features from the input time series data x:
   $$h_{\text{conv}}[i] = \sigma\left(\sum_{j=0}^{k-1} W[j] \cdot x[i+j] + b\right) \tag{4.12}$$
   Here, $h_{\text{conv}}[i]$ denotes the output feature map at position i, σ represents the activation function (e.g., ReLU), W are the learnable filter weights, b is the bias term.

2. **GRU Layer (Gated Recurrent Unit)**:
   Following the convolutional layer, the GRU layer processes the temporal sequence represented by h_{conv}:
   $$h_{\text{rnn}}, \_ = \text{GRU}(h_{\text{conv}}) \tag{4.13}$$
   The GRU updates its hidden states h_{rnn} by selectively incorporating new information while retaining memory of past states.

3. **Recurrent Skip Connection**:
   To capture long-term dependencies, a recurrent skip connection enhances the model's capability:

$$h_{\text{skip\_rnn}}[t] = \text{GRU}(h_{\text{conv}}[t] + h_{\text{skip\_rnn}}[t - \text{skip\_period}]) \tag{4.14}$$

   This mechanism aggregates information across multiple time steps, aiding in the modeling of complex temporal patterns. Here, skip_period denotes the interval for aggregating past information.

4. **Decoder**:
   The decoder module combines outputs from GRU and skip connections to predict target outputs:

$$h_{\text{dec}}, \_ = \text{GRU}([\text{concat}(h_{\text{rnn}}, h_{\text{skip\_rnn}}, h_{\text{skip\_rnn}}[-2:]), x_{\text{initial}}]) \tag{4.15}$$

$$y_{\text{pred}} = \text{FC}(h_{\text{dec}}) \tag{4.16}$$

   Here, y_{pred} represents the model's predicted output, computed via a fully connected layer FC.

The DeepCM model leverages principles from deep learning and time series analysis:

- **Convolutional Operation**:
  Convolutional layers extract local patterns from sequential data using filters, enhancing feature representation:

$$h_{\text{conv}}[i] = \sigma\left(\sum_{j=0}^{k-1} W[j] \cdot x[i+j] + b\right) \tag{4.17}$$

  Here, W denotes the filter weights, b is the bias term, and σ represents the activation function.

- **GRU Mechanism**:
  GRUs facilitate sequence modeling by updating hidden states based on input sequences, maintaining contextual information:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{4.18}$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{4.19}$$

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h) \tag{4.20}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \tag{4.21}$$

- **Recurrent Skip Connection**:
  This connection extends GRU functionality by integrating skip states, capturing periodic dependencies:

$$h_{\text{skip\_rnn}}[t] = \text{GRU}(h_{\text{conv}}[t] + h_{\text{skip\_rnn}}[t - \text{skip\_period}]) \tag{4.22}$$

  The skip period skip_period determines the interval for aggregating past information.

The model undergoes supervised training using LP8 data as input sequences and Sunrise data as target sequences, ensuring accurate prediction through training and evaluation phases.

1. **Data Usage**:
   - Supervised learning model using all Sunrise (high-precision) data as target outputs and all LP8 (low-precision) data as input features.
2. **Training Process**:
   - **Optimizer and Loss Function**:
     The model utilizes the Adam optimizer for parameter optimization and Mean Squared Error (MSE) loss function for model training. During backpropagation, the gradient descent algorithm adjusts model parameters to minimize prediction error.
   - **Dataset Splitting**:
     The dataset is partitioned into training, validation, and testing sets. This partitioning aids in evaluating model performance on unseen data and prevents overfitting. The sliding window technique is employed to handle sequential data, creating overlapping subsets to preserve temporal dependencies.
   - **Detailed Training Steps**:
     - **Data Splitting**: LP8 data as input sequences, Sunrise data as target sequences, segmented using sliding window technique:

$$\text{Input sequence: } X = \{x^{(1)}, x^{(2)}, ..., x^{(m)}\} \tag{4.23}$$

$$\text{Target sequence: } Y = \{y^{(1)}, y^{(2)}, ..., y^{(m)}\} \tag{4.24}$$

     - **Model Initialization**: Initializes the DeepCM model, setting up convolutional layers, GRU layers, recurrent skip connections, and decoder layers:

$$\text{Output prediction: } \hat{Y} = f_{\text{DeepCM}}(X) \tag{4.25}$$

     - **Forward Propagation**: Computes predictions through forward propagation of the model:

$$\hat{Y}^{(t)} = f_{\text{model}}(X^{(t)}) \tag{4.26}$$

     - **Loss Computation**: Computes loss between predicted and target outputs using Mean Squared Error (MSE) loss function:

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 \tag{4.27}$$

- **Backward Propagation**: Updates model parameters by backpropagating through the computed loss:

$$\theta \leftarrow \theta - \alpha \frac{\partial \text{MSE}}{\partial \theta} \tag{4.28}$$

where θ denotes model parameters and α is the learning rate.

- **Training Monitoring**: Prints training loss for each epoch, monitoring the model's training process.

3. **Evaluation**:

   - Evaluates the model using a new test dataset after training completion.
   - Calculates Root Mean Squared Error (RMSE) between calibrated data and original high-precision data as evaluation metric.

In conclusion, the DeepCM model effectively calibrates low-precision sensor data (LP8) to approximate high-precision sensor data (Sunrise), demonstrating its potential in enhancing data accuracy for real-time applications.

4.2.2 Modified Supervised Sensor Calibration

However, if I use all high-cost sensors data and low-cost sensors data to train the model, and use the trained model to calibrate the low-cost sensors, it seems that the calibration has lost its meaning. Since I already have all the high-cost sensor data, I can use it directly without calibrating the low-cost sensors. Therefore, I came up with a new sensor calibration method. Similarly, this calibration method is also based on the DeepCM method. The difference is that I only use 2% of the total data as training data to train the model, and then use the trained model to calibrate the low-cost sensors.

The sensor calibration pseudo-code for the modified DeepCM model is shown in Algorithm 3.

The process of DeepCM Time Series Calibration involves training a Deep Convolutional Memory (DeepCM) model to calibrate low-precision time series data. First, the high-precision and low-precision data are normalized and transformed into a windowed format. The DeepCM model, consisting of an encoder with convolutional and recurrent layers and a decoder with a recurrent layer and a fully connected layer, is then trained on the windowed training data using Mean Squared Error (MSE) loss and an Adam optimizer.

After training, the model is used to predict calibrated values for the entire dataset, window by window. The predicted values are denormalized, and the calibration performance is evaluated by visualizing the calibrated, low-precision, and high-precision data and calculating the Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) between the calibrated and high-precision data.

Algorithm 3 DeepCM Time Series Calibration

Require: Low-precision data (X_l), High-precision data (X_h), window size (w), skip period (p)
Ensure: Calibrated data ($\hat{X}_h$)
Data Preparation
Normalize X_l, X_h
Create windowed datasets: ($X_{l,w}$, $X_{h,w}$)
Model Training
Initialize DeepCM model
Define loss (MSE), optimizer (Adam)
for each epoch **do**
    for each batch ($x_{l,w}$, $x_{h,w}$) in **training data do**
        $h_{enc} \leftarrow$ Encoder($x_{l,w}$)
        $\hat{x}_{h,w} \leftarrow$ Decoder(h_{enc}, $x_{l,w}$[-1])
        $L \leftarrow$ MSE($\hat{x}_{h,w}$, $x_{h,w}$)
        Update model parameters using L
    end for
end for
Calibration
Load trained DeepCM model
for each window in the **entire dataset do**
    $\hat{x}_{h,w} \leftarrow$ Model(normalized window data)
    Denormalize $\hat{x}_{h,w}$
    Collect $\hat{x}_{h,w}$ to form $\hat{X}_h$
end for
Evaluation
Visualize X_l, X_h, and $\hat{X}_h$
Calculate RMSE and MAE between $\hat{X}_h$ and X_h

4.3 Long Short-Term Memory Prediction

4.3.1 Long Short-Term Memory (LSTM) Short-term Prediction

Long Short-Term Memory (LSTM) networks, a specialized form of Recurrent Neural Network (RNN), excel at processing sequential data where long-range dependencies are crucial, such as natural language, speech, and time series. Traditional RNNs, while good at handling sequential data, struggle with vanishing and exploding gradients, limiting their ability to capture long-term dependencies. LSTMs effectively tackle this challenge through their sophisticated gating mechanisms, facilitating the learning and retention of long-range information within sequences.

The flowchart of the LSTM Short-term Prediction model is shown in figure 4.3.1.

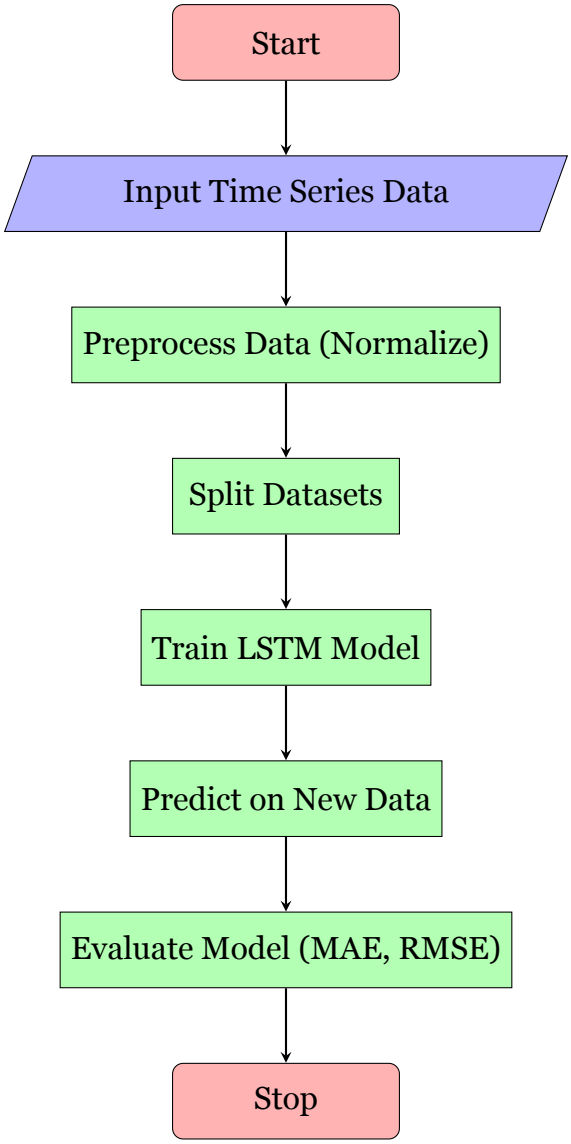


Figure 4.3.1: Flowchart of LSTM Short-term Prediction

This model uses a Long Short-Term Memory (LSTM) neural network to predict time series data through a four-stage process.

First, during data preparation, the input time series is normalized and transformed into a supervised learning problem using a sliding window, creating input-target pairs. Next, an LSTM model with customizable parameters is trained on the windowed training data using the Adam optimizer and MSE loss, while early stopping prevents overfitting. Then, the trained model predicts future values for the test data window by window, with predictions subsequently denormalized to their original scale. Finally, model performance is evaluated by visualizing predicted and actual time series, often alongside error metrics calculation.

The pseudo-code for the sensor short-time prediction model is shown in Algorithm 4.

Algorithm 4 LSTM Time Series Prediction

Require: Time series data (X), window size (w)
Ensure: Predicted time series ($\hat{X}$)
Prepare Data
$X_{norm} \leftarrow$ Normalize(X)
Create windowed training, validation, and test sets from X_{norm}
Train Model
Initialize LSTM model
while Early stopping criterion not met **do**
    Train model on windowed training data using Adam optimizer and MSE loss
    Evaluate model on windowed validation data
end while
Predict
for each window in the test data **do**
    Predict values using trained LSTM model
    Denormalize predictions
    Append predictions to $\hat{X}$
end for
Evaluate
Visualize X and $\hat{X}$

The heart of an LSTM lies in its cell structure, a memory powerhouse with gates controlling information flow. The forget gate discards irrelevant data from the cell state. The input gate decides what new information to store, utilizing a tanh layer to regulate additions. Finally, the output gate, influenced by the cell state and current input, determines which information to output, passing it to the next time step. This intricate dance of gates allows LSTMs to learn long-term dependencies in sequential data.

1. Cell State (C_t)

The cell state in an LSTM acts as a memory bank, carrying information throughout the entire LSTM chain. It is updated dynamically to maintain long-term dependencies.

The update formula for the cell state is:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{4.29}$$

where:

- C_{t-1} is the cell state from the previous timestep.
- f_t is the forget gate vector, determining how much of the previous cell state to retain.
- i_t is the input gate vector, determining how much new information to add to the cell state.

- $\tilde{C}_t$ is the candidate cell state, computed based on the input and the previous hidden state.
- $\odot$ denotes element-wise multiplication.

The candidate cell state $\tilde{C}_t$ is computed as:

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{4.30}$$

2. Hidden State (h_t)

The hidden state in an LSTM represents the output of the unit at each timestep, carrying information forward. It is determined based on the current cell state and the output gate.

The formula for the hidden state is:

$$h_t = o_t \odot \tanh(C_t) \tag{4.31}$$

where:

- h_t is the hidden state at the current timestep.
- o_t is the output gate vector, determining which parts of the cell state influence the hidden state output.

The output gate o_t is computed as:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{4.32}$$

In these formulas:

- x_t is the input at the current timestep.
- h_{t-1} is the hidden state from the previous timestep.
- W and b are the weight matrices and bias vectors associated with each operation.
- σ denotes the sigmoid activation function.
- tanh denotes the hyperbolic tangent activation function.

3. Gating Mechanisms

These regulatory gates govern the flow and modification of information, enabling LSTMs to circumvent the gradient pitfalls of RNNs and achieve long-term memory. There are three main gates:

- **Input Gate (i_t):** This gate determines what new information deserves entry into the cell state.
  - **Input Gate Value:** Governed by the equation:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{4.33}$$

    where:

    - W_i: Weight matrix for the input gate.
    - h_{t-1}: Hidden state from the previous timestep.
    - x_t: Current input.
    - b_i: Bias vector for the input gate.
    - σ: Sigmoid function, compressing values between 0 and 1, representing the gate's "openness."

- **Forget Gate (f_t):** Responsible for deciding which information to discard from the cell state, this gate employs the following equation:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{4.34}$$

  where:

  - W_f: Weight matrix for the forget gate.
  - b_f: Bias vector for the forget gate.

- **Output Gate (o_t):** This gate regulates which parts of the cell state are read and outputted to the hidden state, using the equation:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{4.35}$$

  where:

  - W_o: Weight matrix for the output gate.
  - b_o: Bias vector for the output gate.

The computational flow within an LSTM unit follows these steps:

1. The Forgetting Ritual:

The forget gate decides what to discard from the previous cell state:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{4.36}$$

2. Embracing New Information:

The input gate determines the extent to which new information is integrated:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{4.37}$$

Simultaneously, a candidate cell state is prepared, representing potential new information to store:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{4.38}$$

where:

- W_C: Weight matrix for the candidate cell state.
- b_C: Bias vector for the candidate cell state.
- tanh: Hyperbolic tangent function, compressing values between -1 and 1.

3. The Cell State's Evolution:

The cell state is then updated, blending old and new information:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{4.39}$$

This step embodies the essence of long-term memory. The forget gate governs the retention of old information ($f_t * C_{t-1}$), while the input gate controls the integration of new information ($i_t * \tilde{C}_t$).

4. Outputting Refined Information:

The output gate determines what to extract from the cell state:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{4.40}$$

5. The Hidden State's Genesis:

Finally, the hidden state for the current timestep is generated:

$$h_t = o_t * \tanh(C_t) \tag{4.41}$$

After understanding the basic structure and calculation process of LSTM, I will continue to introduce the specific application of LSTM in short-term predicting. Next, I will introduce in detail the process and details of my model using LSTM for short-term prediction.

This model implements a time series prediction model based on Long Short-Term Memory (LSTM) networks, covering data preprocessing, model training, evaluation, and prediction. Here's a detailed description of its process:

1. Data Pre-process

Before training the model, the data undergoes preprocessing steps to ensure it is suitable for input into the LSTM network. This includes normalization of the data and formatting it into the appropriate tensor format.

Normalization's primary goal is to scale numerical data to a specific range, enhancing model training stability and convergence speed. A common normalization technique is MinMax scaling, where data is transformed to fit within a predefined range, typically $[-1, 1]$ or $[0, 1]$.

The MinMax scaling formula is:

$$X_{\text{scaled}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \times (\text{max} - \text{min}) + \text{min} \tag{4.42}$$

Where:

- X represents the original value of the feature.
- X_{min} is the minimum value of the feature in the dataset.

- X_{max} is the maximum value of the feature in the dataset.
- min and max denote the desired range (e.g., -1 and 1 for $[-1, 1]$).

Once the data is normalized, it needs to be transformed into a format suitable for input into an LSTM model. LSTM models in frameworks like TensorFlow or PyTorch typically expect inputs in the form of tensors (multi-dimensional arrays). The key steps in data format transformation for LSTM inputs include:

2. Model Training

The LSTM model is defined and trained using the preprocessed data. This section details the training process, including model architecture, loss function selection, optimizer choice, and training iterations.

- **Model Definition**: A neural network model comprising multiple LSTM layers and fully connected layers is designed to learn patterns from time series data. The structure of LSTM has been described in detail above, so I will not repeat it here.
- **Loss Function and Optimizer**:

  Appropriate loss functions (e.g., Mean Squared Error) measure discrepancies between predicted and actual values. Optimizers (e.g., Adam optimizer) adjust model weights to minimize the loss function, enhancing model fitting to the data.

  Loss functions quantify the discrepancy between predicted and actual values in machine learning models.

  Mean Squared Error is a common loss function for regression tasks:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.43}$$

  Binary Cross-Entropy measures the performance of binary classification models:

$$\text{Binary Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{4.44}$$

  Multiclass Cross-Entropy assesses the accuracy of multiclass classification models:

$$\text{Multiclass Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic}) \tag{4.45}$$

  Optimizers adjust model parameters to minimize the chosen loss function during training.

  Stochastic Gradient Descent updates model parameters based on the gradient of the loss function:

$$\theta_{t+1} = \theta_t - \eta\nabla_\theta L(\theta) \tag{4.46}$$

Adam optimizer combines adaptive learning rates and momentum for efficient gradient-based optimization:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{4.47}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{4.48}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} \cdot m_t \tag{4.49}$$

- **Training Process**: During the training process of a machine learning model, several key steps are undertaken to optimize its parameters and improve its performance on the dataset. Below is a structured outline of these steps:

  (a) Initialization:

  Before training begins, model parameters such as weights W and biases b are initialized to some initial values $W^{(0)}, b^{(0)}$.

  $$W^{(0)}, b^{(0)} \text{ (initial values)} \tag{4.50}$$

  (b) Forward Pass:

  In each epoch, the model computes predicted outputs $\hat{y}_i$ for each data point x_i in the training dataset using the current parameters $W^{(t)}, b^{(t)}$.

  $$\hat{y}_i = f(x_i; W^{(t)}, b^{(t)}) \tag{4.51}$$

  Here, f represents the model's forward pass function at epoch t.

  (c) Loss Calculation:

  The loss L is computed to quantify the difference between predicted outputs $\hat{y}_i$ and actual targets y_i using a specified loss function $\mathcal{L}$.

  $$L(W^{(t)}, b^{(t)}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, \hat{y}_i) \tag{4.52}$$

  Here, N denotes the number of training samples.

  (d) Backward Pass (Backpropagation):

  Gradients of the loss function with respect to model parameters $W^{(t)}$ and $b^{(t)}$ are calculated using backpropagation.

  $$\nabla_{W^{(t)}} L = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}(y_i, \hat{y}_i)}{\partial W^{(t)}}, \quad \nabla_{b^{(t)}} L = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}(y_i, \hat{y}_i)}{\partial b^{(t)}} \tag{4.53}$$

  (e) Gradient Descent (Parameter Update):

  Using the gradients computed in the previous step, model parameters $W^{(t+1)}$ and $b^{(t+1)}$ are updated to minimize the loss, scaled by a learning rate η.

  $$W^{(t+1)} = W^{(t)} - \eta \cdot \nabla_{W^{(t)}} L, \quad b^{(t+1)} = b^{(t)} - \eta \cdot \nabla_{b^{(t)}} L \tag{4.54}$$

(f) Validation:

Periodically, the model's performance is evaluated on a validation dataset using the same loss function $\mathcal{L}$ to monitor its ability to generalize to unseen data.

(g) Iterative Process:

Steps (a)-(f) are repeated for a fixed number of epochs or until convergence criteria are met, allowing the model to learn progressively from the training data.

3. Model Evaluation and Prediction

Once trained, the LSTM model is evaluated using a separate test dataset to assess its performance in predicting future values. This section discusses how the model's accuracy is evaluated and how predictions are made.

After training, the LSTM model's performance is assessed using a separate test dataset. This evaluation involves:

- **Error Calculation**: Comparing the model's predictions with actual values from the test dataset using metrics such as Mean Squared Error (MSE) or Mean Absolute Error (MAE).

Once evaluated, the LSTM model is used for making predictions:

- **New Input Data**: Takes new input data, typically representing future time steps.
- **Predicting**: Uses learned patterns to predict future values beyond the trained data.
- **Output**: Provides predictions for upcoming time steps based on the input data.

These steps ensure that the LSTM model can effectively predict future values based on its training and generalize well to unseen data, making it useful for predicting applications.

4.3.2 Long Short-Term Memory Encoder-Decoder (LSTM-ED) Short-term Prediction

4.3.2.1 Single Sensor LSTM-ED Short-term Prediction

1. Encoder

An LSTM unit is a specialized type of recurrent neural network (RNN) that effectively captures dependencies over long sequences by maintaining a cell state. The LSTM unit comprises several components:

- Input gate: i_t
- Forget gate: f_t

- Output gate: o_t
- Cell state: c_t
- Hidden state: h_t

These components work together to control the flow of information and maintain long-term dependencies.

The Encoder typically consists of multiple stacked LSTM layers. Each layer processes the input sequence step-by-step, updating its hidden state and cell state at each time step.

For the first layer ($l = 1$), the LSTM operation is given by:

$$h_t^{(1)}, c_t^{(1)} = \text{LSTM}_1(x_t, h_{t-1}^{(1)}, c_{t-1}^{(1)}) \tag{4.55}$$

For subsequent layers ($l > 1$), the operation is:

$$h_t^{(l)}, c_t^{(l)} = \text{LSTM}_l(h_t^{(l-1)}, h_{t-1}^{(l)}, c_{t-1}^{(l)}) \tag{4.56}$$

Here, LSTM_l denotes the LSTM operation in the l-th layer, and $h_t^{(l-1)}$ is the hidden state from the previous layer at time step t.

To better understand the LSTM operations, I will now explain the relationships and significance of these formulas in detail. Each formula plays a specific role in managing the flow of information through the LSTM unit:

1. Input Gate: The input gate (i_t) decides how much of the new input (x_t) and the previous hidden state (h_{t-1}) should be used to update the cell state (c_t).

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{4.57}$$

2. Forget Gate: The forget gate (f_t) determines the extent to which the previous cell state (c_{t-1}) should be carried forward. This gate helps in selectively forgetting parts of the cell state.

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{4.58}$$

3. Output Gate: The output gate (o_t) controls how much of the cell state should be used to compute the hidden state (h_t). The hidden state is what gets passed to the next time step and potentially to the next layer.

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{4.59}$$

4. Candidate Cell State: The candidate cell state ($\tilde{c}_t$) is a potential update to the cell state based on the new input and the previous hidden state.

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{4.60}$$

5. Cell State Update: The cell state (c_t) is updated by combining the old cell state (c_{t-1}) and the candidate cell state ($\tilde{c}_t$), modulated by the forget gate (f_t) and input gate (i_t). This step is crucial for maintaining long-term dependencies.

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{4.61}$$

6. Hidden State Update: The hidden state (h_t) is updated based on the current cell state (c_t) and the output gate (o_t). The hidden state represents the output of the LSTM unit and is used for predictions and passed to the next time step.

$$h_t = o_t \odot \tanh(c_t) \tag{4.62}$$

In these equations, σ is the sigmoid function, $\odot$ denotes element-wise multiplication, W_i, W_f, W_o, W_c are input weight matrices, U_i, U_f, U_o, U_c are recurrent weight matrices, and b_i, b_f, b_o, b_c are bias vectors.

The primary role of the Encoder is to process the input time series data $\{x_1, x_2, ..., x_T\}$ and encode it into a fixed-dimensional hidden representation. This representation captures the temporal dependencies and is essential for generating accurate predicts in the subsequent decoding phase.

By sequentially processing the input sequence and updating hidden and cell states, the Encoder forms the foundation for meaningful representations that drive accurate predictions.

2. Decoder

Following the Encoder, the Decoder in a sequence-to-sequence model mirrors the layer configuration of the Encoder. It is composed of multiple LSTM (Long Short-Term Memory) layers matching the depth of the Encoder. The purpose of the Decoder is to utilize the context provided by the Encoder to generate future values in a sequence.

The Decoder uses LSTM layers to handle temporal dependencies in the data. Each LSTM layer in the Decoder can be represented by a set of equations. Below, I will describe each equation in detail.

1. Input Gate (i_t)

The input gate controls the extent to which a new value flows into the cell state. It is calculated using the following equation:

$$i_t = \sigma(W_i[h_{t-1}, y_{t-1}] + b_i) \tag{4.63}$$

- i_t: Input gate vector at time step t.
- σ: Sigmoid activation function.
- W_i: Weight matrix for the input gate.
- h_{t-1}: Hidden state vector from the previous time step $t-1$.

- y_{t-1}: Previous output from the Decoder at time step $t-1$.
- b_i: Bias vector for the input gate.

2 .Forget Gate (f_t)

The forget gate determines which parts of the previous cell state are carried forward. The forget gate is calculated using the following equation:

$$f_t = \sigma(W_f[h_{t-1}, y_{t-1}] + b_f) \tag{4.64}$$

- f_t: Forget gate vector at time step t.
- W_f: Weight matrix for the forget gate.
- b_f: Bias vector for the forget gate.

3. Output Gate (o_t)

The output gate decides which part of the cell state will be outputted. It is calculated using the following equation:

$$o_t = \sigma(W_o[h_{t-1}, y_{t-1}] + b_o) \tag{4.65}$$

- o_t: Output gate vector at time step t.
- W_o: Weight matrix for the output gate.
- b_o: Bias vector for the output gate.

The candidate cell state represents new information that could be added to the cell state. It is calculated using the following equation:

$$\tilde{C}_t = \tanh(W_C[h_{t-1}, y_{t-1}] + b_C) \tag{4.66}$$

- $\tilde{C}_t$: Candidate cell state at time step t.
- tanh: Hyperbolic tangent activation function.
- W_C: Weight matrix for the candidate cell state.
- b_C: Bias vector for the candidate cell state.

5. Cell State (C_t)

The cell state is updated by combining the previous cell state, scaled by the forget gate, with the candidate cell state, scaled by the input gate. The cell state is calculated using the following equation:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{4.67}$$

- C_t: Cell state at time step t.

- C_{t-1}: Cell state from the previous time step.
- $*$: Element-wise multiplication.

6. Hidden State (h_t)

The hidden state is the filtered version of the cell state, controlled by the output gate and transformed by a tanh function. It is calculated using the following equation:

$$h_t = o_t * \tanh(C_t) \tag{4.68}$$

- h_t: Hidden state at time step t.

The Decoder is initialized with the final hidden state and cell state from the Encoder. The initialization is performed as follows:

$$h_0^{\text{decoder}} = h_T^{\text{encoder}} \tag{4.69}$$
$$C_0^{\text{decoder}} = C_T^{\text{encoder}} \tag{4.70}$$

Where T is the final time step in the Encoder. These initial states provide the context from the entire input sequence processed by the Encoder.

At each time step t, the Decoder predicts the next value in the sequence. This prediction is based on the previous output and the hidden state. The process can be described in the following steps:

1. **Input Feeding:** The Decoder takes the previous output y_{t-1} (or a start token for $t = 0$) and combines it with the hidden state h_{t-1}.
2. **LSTM Processing:** The combined input is fed through the LSTM layers, which update their hidden states and cell states according to the equations defined above.
3. **Prediction:** The final hidden state h_t of the top LSTM layer is passed through a dense layer (with a softmax activation if predicting categorical outputs) to produce the next output y_t.

The prediction is made using a dense layer with a softmax activation function, as shown below:

$$y_t = \text{softmax}(W_s h_t + b_s) \tag{4.71}$$

- y_t: Output at time step t.
- W_s: Weight matrix for the dense layer.
- b_s: Bias vector for the dense layer.

The softmax function is used to convert the output into a probability distribution when predicting categorical values.

The Decoder continues this process iteratively to generate the entire sequence of future values. The accuracy of the sequence predicting depends on the quality of the context

vector passed from the Encoder and the ability of the Decoder to leverage this context effectively at each time step.

In summary, the Decoder in a sequence-to-sequence model utilizes multiple LSTM layers, initialized with the final states from the Encoder, to predict future values in a sequence by iteratively processing previous outputs and hidden states. This configuration ensures the Decoder can effectively leverage the temporal dependencies captured by the Encoder, facilitating accurate sequence predicting.

3. Fully Connected Layer

In an LSTM network, the decoder and the fully connected layer are closely related components. The decoder is part of a sequence-to-sequence (Seq2Seq) model tasked with converting the context vectors or hidden state vectors generated by the encoder into a target sequence. The fully connected layer, on the other hand, plays a critical role in refining the outputs of the decoder into precise predictions.

In the following part, I will delve into how the LSTM decoder and the fully connected layer interact to produce accurate predictions.

The decoder's primary function is to transform the context vector or hidden state sequence produced by the encoder into a target sequence. For tasks such as time series prediction or natural language processing, the decoder generates outputs step by step, typically as high-dimensional vectors containing information for each time step.

Once the LSTM decoder generates its outputs, the fully connected layer (also known as a dense layer) comes into play. It receives the output from each time step of the decoder and transforms it into specific values in the target prediction space. Through linear transformations and possibly an activation function, it maps the high-dimensional output from the decoder to the final predicted results.

Next, I will define the relationships that govern the interaction between the LSTM decoder and the fully connected layer.

The hidden state vector $\mathbf{h}_t$ generated by the decoder at time step t undergoes transformation by the fully connected layer:

$$\mathbf{z}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b} \tag{4.72}$$

Here:

- $\mathbf{h}_t$ is the hidden state vector of the decoder at time step t.
- $\mathbf{W}$ is the weight matrix that maps the decoder's output to the target space.
- $\mathbf{b}$ is the bias vector.
- $\mathbf{z}_t$ is the intermediate vector before applying the activation function.

The activation function f is then applied to $\mathbf{z}_t$ to obtain the final prediction $\mathbf{y}_t$:

$$\mathbf{y}_t = f(\mathbf{z}_t) \tag{4.73}$$

Common activation functions include ReLU, sigmoid, or tanh.

The process starts with the encoder generating a context vector $\mathbf{c}$ or a sequence of hidden states based on the input sequence. This information is then passed to the decoder. Once the decoder receives the encoded information, it generates the hidden state vectors $\mathbf{h}_t$ for each time step. These vectors are subsequently passed to the fully connected layer. The fully connected layer performs a linear transformation on $\mathbf{h}_t$, producing the intermediate vector $\mathbf{z}_t$. An activation function f is applied to $\mathbf{z}_t$ to obtain the final prediction $\mathbf{y}_t$.[72]

In summary, the fully connected layer in the decoder plays a crucial role in transforming the high-dimensional hidden state vectors produced by the LSTM decoder into precise predictions. Through linear transformations and activation functions, it maps complex time-series features to concrete prediction values, ensuring accurate sequence-to-sequence predictions. This close collaboration ensures that the model exhibits robust predictive capabilities when handling complex tasks.

4. Workflow

This model implements an Encoder-Decoder LSTM model for time series prediction. The pseudo-code of the model is shown in Algorithm 5. The model uses two LSTM networks: an encoder to process the input sequence and a decoder to generate the output sequence.

During training, the input time series is normalized and transformed into a sequence of differences between consecutive data points. The encoder LSTM processes sliding windows of this preprocessed data and produces hidden and cell states. These states capture the temporal dependencies within the window. The decoder LSTM then takes these states as input and generates predictions.

The model is trained using the Adam optimizer and Mean Squared Error (MSE) loss. Early stopping is employed to prevent overfitting by monitoring the validation loss. Once trained, the model can predict future values for a given window of data and the desired prediction horizon. The predicted values are then denormalized to obtain the final predictions in the original data scale.

Algorithm 5 Encoder-Decoder LSTM for Time Series Prediction

Require: Time series data (X), window size (w), target length (t)
Ensure: Predicted time series ($\hat{X}$)
Data Preprocessing
$X_{diff} \leftarrow$ Normalize and difference(X)
Model Training
Initialize encoder-decoder LSTM model
while Early stopping criterion not met **do**
    for each window in training data **do**
        $h, c \leftarrow$ Encoder(window data)
        $\hat{X}_{window} \leftarrow$ Decoder(h, c, target length t)
        Update model parameters using backpropagation
    end for
    Evaluate model on validation data
end while
Prediction
for each window in the data **do**
    $\hat{X}_{window} \leftarrow$ Predict next t values using trained model and current window
    $\hat{X}$.append($\hat{X}_{window}$[-1])
end for
Denormalize $\hat{X}$
Evaluation
Visualize X and $\hat{X}$

In the modeling workflow, I will emphasize rigorous data preprocessing, effective model training, thorough evaluation, and accurate prediction to ensure robust performance in predicting tasks. The flowchart of the LSTM Encoder-Decoder model is shown in figure 4.3.2.

1. Data Preprocessing

Data preprocessing is a critical preparatory step in our model workflow. It involves two key techniques: normalization and differencing. Normalization using MinMaxScaler scales data to a range of -1 to 1, enhancing model stability. Differencing further extracts trend information from the normalized data, essential for mitigating seasonal or trend effects within the time series.

2. Model Training

Once preprocessed, the data is split into training and validation sets. The model undergoes training using the training set, with performance monitored using the validation set. Early stopping strategies are employed to prevent overfitting and optimize model convergence based on validation metrics.

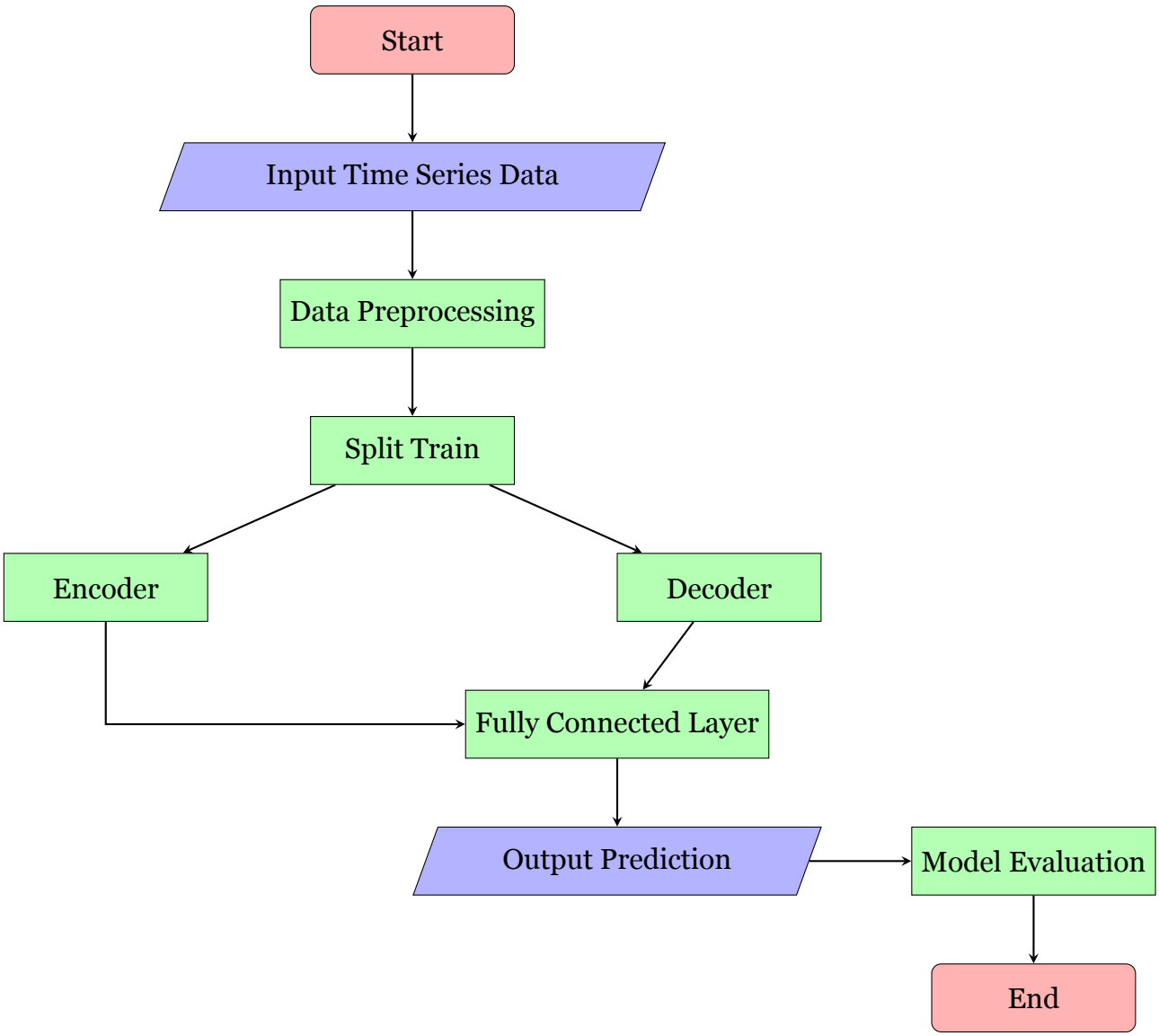


Figure 4.3.2: Flowchart of LSTM Encoder-Decoder Short-term Prediction

In training process, the first step in model training is data splitting. This ensures that the data is divided into two main subsets:

- **Training Set:** This subset is used to train the model. It consists of a large portion of the available data and serves as the basis for the model to learn patterns and relationships between input features and target outputs.
- **Validation Set:** The validation set is crucial for evaluating the model during training. It helps monitor the model's performance on unseen data and guides adjustments to hyperparameters to prevent overfitting.

Once the data is split, the model undergoes training using the training set. This involves iterative optimization of model parameters (θ) to minimize a chosen loss function ($L(\theta)$):

$$\theta^* = \arg\min_{\theta} L(\theta) \tag{4.74}$$

During training, optimization methods such as stochastic gradient descent (SGD) adjust model parameters based on gradients of the loss function:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta L(\theta^{(t)}) \tag{4.75}$$

where η is the learning rate and $\nabla_\theta L(\theta^{(t)})$ is the gradient of the loss function with respect to θ.

Validation and early stopping strategies are integral to effective model training. Validation metrics are computed using the validation set to monitor the model's performance. Early stopping strategies prevent overfitting by halting the training process if the validation performance does not improve for a specified number of epochs (iterations).

3. Prediction

When using LSTM Encoder-Decoder model for time series predicting, generating predictions for future time steps involves several critical steps. Once the model is trained and validated, it can be used to predict future values.

The trained model outputs predictions typically in the scaled or differenced format of the data. For instance, if the data were normalized to a range like [0, 1] or standardized, the model's predictions would also be in this scaled range.

To translate these predictions back to the original data scale for practical use, inverse normalization and differencing steps are applied.

Differencing is a common preprocessing technique used to stabilize non-stationary time series data by computing the differences between consecutive observations. If the original time series $\{y_1, y_2, ..., y_T\}$ was differenced, resulting in $\{y_2 - y_1, y_3 - y_2, ..., y_T - y_{T-1}\}$, the inverse differencing operation to obtain the original scale prediction $\hat{y}_{\text{original},t}$ from the model's predicted differenced value $\hat{y}_t$ is:

$$\hat{y}_{\text{original},t} = \hat{y}_t + y_{\text{last}} \tag{4.76}$$

where:

- $\hat{y}_{\text{original},t}$ is the original scale prediction at time step t.
- $\hat{y}_t$ is the model's predicted differenced value.
- y_{last} is the last observed value from the original time series, used to reverse the differencing operation.

Normalization scales data to a predefined range (e.g., [0, 1] or [-1, 1]), which accelerates training and ensures stability in deep learning models. If the original data $\{y_1, y_2, ..., y_T\}$ was normalized, the inverse normalization to convert the model's normalized prediction $\hat{y}_{\text{norm}}$ back to the original scale $\hat{y}_{\text{final}}$ is:

$$\hat{y}_{\text{final}} = \hat{y}_{\text{norm}} \cdot (\max - \min) + \min \tag{4.77}$$

where:

- $\hat{y}_{\text{final}}$ is the final prediction in the original data scale.
- $\hat{y}_{\text{norm}}$ is the model's predicted value in the normalized range.
- max and min are the maximum and minimum values of the original data, which were used during normalization.

These steps are essential not only for ensuring the interpretability of predictions but also for integrating deep learning models effectively into practical decision-making processes involving time series data.

4. Model Evaluation

In this model, evaluating the performance of a trained regression model involves several metrics that quantify the difference between predicted values and actual values in the test dataset. Two commonly used metrics are Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE).

In regression analysis, Mean Absolute Error (MAE) is a metric that measures the average absolute difference between predicted values and actual values. It is less sensitive to outliers compared to RMSE because it does not square the differences.

MAE is calculated as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4.78}$$

where:

- y_i is the actual value of the target variable for instance i,
- $\hat{y}_i$ is the predicted value of the target variable for instance i,
- n is the number of instances in the test dataset.

MAE provides a measure of the average absolute deviation between predictions and actual observations.

RMSE is another widely used metric in regression tasks, which measures the square root of the average of squared differences between predicted values and actual values. It penalizes larger errors more heavily than MAE due to squaring the differences.

RMSE is calculated as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{4.79}$$

where:

- y_i is the actual value of the target variable for instance i,

- $\hat{y}_i$ is the predicted value of the target variable for instance i,
- n is the number of instances in the test dataset.

RMSE provides a measure of the average magnitude of the error made by the model in its predictions.

Both MAE and RMSE are crucial metrics for evaluating the performance of regression models:

- **MAE**: Useful when the absolute magnitude of errors is important and outliers should not be excessively penalized.
- **RMSE**: Provides a more comprehensive picture by considering both the variance and the magnitude of errors, making it more suitable when large errors should be significantly penalized.

Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) are fundamental metrics in regression analysis. They provide insights into the accuracy and performance of models on unseen data, aiding in model comparison, parameter tuning, and overall model improvement strategies.

These metrics are widely applicable across various domains of machine learning and play a crucial role in ensuring the reliability and effectiveness of regression models in real-world applications.

4.3.2.2 Topological Relationship Analysis

Sensor networks rely on understanding the interconnections between sensors to optimize data collection and analysis. This section explores the fundamental principles and methodologies involved in analyzing the topology of sensor networks.[73]

1. Correlation Analysis

Understanding the correlation between sensor data is essential for identifying relationships influenced by common environmental factors.

The Pearson correlation coefficient quantifies the linear relationship between two variables X and Y:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{4.80}$$

where $cov(X,Y)$ is the covariance, σ_X and σ_Y are standard deviations, and μ_X and μ_Y are means. The coefficient ranges from $[-1,1]$, indicating stronger correlations near ± 1.

Sensors with higher correlation coefficients likely share proximity or monitor similar pollutants. Finally I can get the sensor correlation coefficient matrix graph as shown in figure.

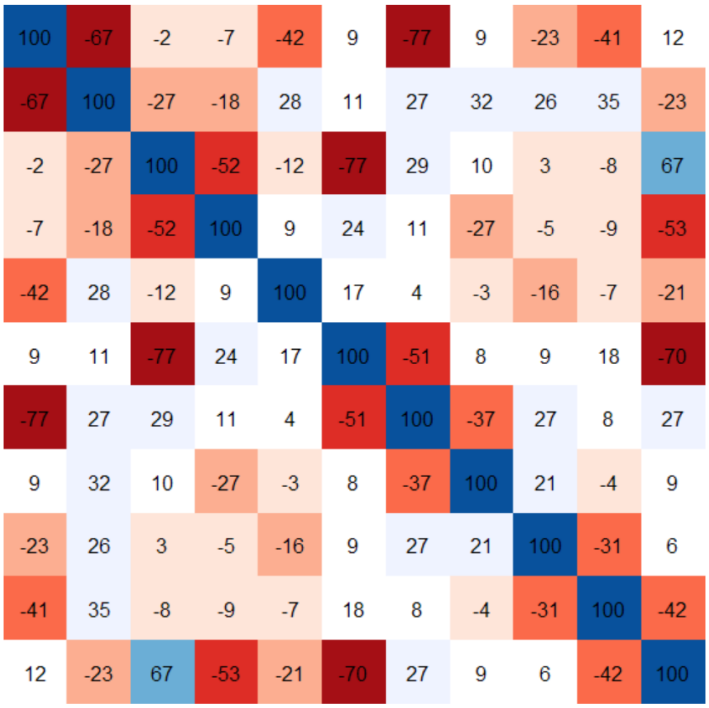


Figure 4.3.3: Correlation Coefficient Matrix

2. Distance Factor

Geographical proximity plays a crucial role in sensor data correlation, significantly impacting both network design and data interpretation. Sensors located close to each other tend to experience similar environmental influences, and this principle can be applied practically by establishing a distance threshold $d_{\max}$. This threshold helps identify sensors that are potentially correlated due to their proximity, simplifying data analysis and improving the efficiency of network design.

Graph theory provides a robust framework for modeling sensor networks, where nodes represent sensors and edges denote connections. This approach offers valuable insights into network structure, facilitating fault detection and optimizing data routing. By constructing a topology graph, the interconnections among sensors are visually represented, which aids in maximizing network performance.

The optimal deployment of sensors relies on systematic data collection and thorough correlation analysis, ensuring effective utilization of network resources and accurate environmental monitoring. Precise data acquisition is critical for accurate network analysis. The process begins by collecting essential data from each sensor, including geographical coordinates (x_i, y_i) and gas concentration readings. This data forms the foundation for subsequent analysis.

To quantify relationships between sensors, the correlation coefficient between any two sensors i and j is computed. The correlation coefficient $\rho_{i,j}$ is given by:

$$\rho_{i,j} = \frac{\sum_{k=1}^{n}(X_{i,k} - \bar{X}_i)(X_{j,k} - \bar{X}_j)}{\sqrt{\sum_{k=1}^{n}(X_{i,k} - \bar{X}_i)^2 \sum_{k=1}^{n}(X_{j,k} - \bar{X}_j)^2}}, \tag{4.81}$$

where $X_{i,k}$ and $X_{j,k}$ represent gas concentration measurements from sensors i and j at time k, and $\bar{X}_i$, $\bar{X}_j$ are their respective mean values. The result is then used to construct a correlation matrix **R**, where each entry $R_{i,j} = |\rho_{i,j}|$ highlights the strength of the correlation between sensors.

Once correlations are established, sensor relationships can be mapped using graph theory. We define the graph $G = (V, E)$, where V represents the set of sensors as nodes, and E represents the edges between them. Two sensors are connected by an edge if they meet predefined proximity and correlation thresholds, representing both geographical and data similarity.

Finally, graphical representations of these relationships provide deeper insights for network optimization. Heatmaps of the correlation matrix **R** visualize the strength of data correlations, while network plots of G display sensor connections, helping to better understand and manage the sensor network.

The topology of a sensor network is delineated based on geographical proximity, correlation analysis, and graph theory principles. By integrating these methodologies, the network can be structured to optimize data collection, enhance fault detection, and facilitate efficient data routing. Understanding the spatial and functional relationships between sensors ensures robust performance and informed decision-making in sensor network deployment and management.

4.3.2.3 Muti-sensor LSTM Encoder-Decoder Short-term Prediction

After completing the analysis of the sensor topology, I hope to explore a new short-term prediction method based on the LSTM encoder-decoder. I hope to perform sensor fusion through the sensor topology obtained through analysis, and use the LSTM encoder-decoder model for short-term prediction during the fusion process to improve the prediction accuracy.

The multi-sensor LSTM Encoder-Decoder model is not much different from the single-sensor LSTM Encoder-Decoder model, because both are short-term prediction models based on the LSTM encoder-decoder. However, there are some differences between the two in other aspects such as input data, so I will not introduce this model in detail, but will compare and analyze the differences between the two models.

1. Data Input

(a) Single-Sensor LSTM Encoder-Decoder Model

This model processes univariate time series data from a single sensor.

- **Input Data**: Univariate time series $\{x_1, x_2, \ldots, x_T\}$.
- **Features**: Contains information from a single sensor.

(b) Multi-Sensor Fused LSTM Encoder-Decoder Model

This model handles multivariate time series data, integrating information from multiple sensors.

- **Input Data**: Multivariate time series $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$, where $\mathbf{x}_t = [x_t^{(1)}, x_t^{(2)}, \ldots, x_t^{(N)}]$.
- **Features**: Includes data from multiple sensors.

2. Information Processing

(a) Single-Sensor LSTM Encoder-Decoder Model

This model processes each time step individually from a single time series. It is designed for handling univariate time series data from a single sensor.

$$h_t, c_t = \text{LSTM}(x_t, h_{t-1}, c_{t-1}) \tag{4.82}$$

$$\hat{x}_{t+1}, h_t, c_t = \text{LSTM}(\hat{x}_t, h_{t-1}, c_{t-1}) \tag{4.83}$$

In the single-sensor model:

- x_t represents the input at time step t from the single sensor.
- $\hat{x}_{t+1}$ denotes the predicted output at time step $t+1$.
- h_t and c_t are the hidden and cell states of the LSTM at time step t, respectively. These states capture the temporal dependencies and patterns in the single-sensor data.

(b) Multi-Sensor Fused LSTM Encoder-Decoder Model

This model integrates information from multiple sensors at each time step. It is suitable for handling multivariate time series data where each time step includes data from multiple sensors.

$$h_t, c_t = \text{LSTM}(\mathbf{x}_t, h_{t-1}, c_{t-1}) \tag{4.84}$$

$$\hat{\mathbf{x}}_{t+1}, h_t, c_t = \text{LSTM}(\hat{\mathbf{x}}_t, h_{t-1}, c_{t-1}) \tag{4.85}$$

In the multi-sensor model:

- $\mathbf{x}_t = [x_t^{(1)}, x_t^{(2)}, \ldots, x_t^{(N)}]$ represents the input vector at time step t from N sensors. Each element $x_t^{(i)}$ corresponds to the measurement from sensor i.
- $\hat{\mathbf{x}}_{t+1} = [\hat{x}_{t+1}^{(1)}, \hat{x}_{t+1}^{(2)}, \ldots, \hat{x}_{t+1}^{(N)}]$ denotes the predicted output vector at time step $t+1$ for each sensor.
- h_t and c_t are the hidden and cell states of the LSTM at time step t. These states incorporate information from all sensors, enabling the model to learn relationships and dependencies across different sensor measurements.

The multi-sensor fused model enhances the capability to capture complex interactions and dependencies between sensors, which can be crucial for applications requiring comprehensive analysis of multivariate time series data.

3. Model Complexity

(a) Single-Sensor LSTM Encoder-Decoder Model

This model processes each time step individually from a single time series. It is designed to learn temporal dependencies and predict future values based solely on the history of a single sensor's data.

- **Parameters**: The model has fewer parameters compared to the multi-sensor model. It primarily involves the LSTM parameters (weights and biases) and the linear layer for decoding.
- **Training Time**: Due to handling a single time series, the training time is generally shorter compared to models that process multiple sensors simultaneously.
- **Model Complexity**: The model complexity is lower because it deals with a univariate input, focusing on learning temporal dependencies within a single sensor's data. This simplicity can lead to faster convergence during training and potentially less overfitting when data is limited.

(b) Multi-Sensor Fused LSTM Encoder-Decoder Model

This model integrates information from multiple sensors at each time step. It is designed to capture complex relationships and interactions between different sensors' data over time.

- **Parameters**: The model has more parameters compared to the single-sensor model. It involves learning relationships between multiple sensors, resulting in a higher number of LSTM parameters and possibly additional parameters for fusion mechanisms.
- **Training Time**: Due to the complexity of handling multiple sensors and their interactions, the training time is generally longer compared to the single-sensor model.
- **Model Complexity**: The model complexity is higher because it must capture complex interactions and dependencies between multiple sensors over time. This involves learning to fuse information from different sources effectively, which can lead to better representation of the underlying dynamics but also requires more data and careful regularization to avoid overfitting.

The choice between these models depends on the specific requirements of the application. The single-sensor model offers simplicity and efficiency when dealing with univariate time series data, suitable for tasks where data from a single source provides sufficient predictive power. In contrast, the multi-sensor fused model offers enhanced capability to leverage diverse data sources, making it advantageous for applications requiring comprehensive analysis and prediction based on multiple interrelated signals.

4. Suitable Scenarios

(a) Single-Sensor LSTM Encoder-Decoder Model

This model is suitable for tasks requiring prediction based on a single sensor's data.

Applications: Suitable for single-type data time series prediction, such as temperature predicting from a single weather sensor.

Advantages: The model's simplicity and fast training speed are advantageous for applications where handling a single time series efficiently is critical. It focuses on learning temporal dependencies within a single source of data.

(b) Multi-Sensor Fused LSTM Encoder-Decoder Model

This model is suitable for tasks requiring integration of data from multiple sensors.

Applications: Suitable for applications involving integration of data from diverse sources, such as smart city systems combining data from traffic sensors, weather stations, and pollution monitors.

Advantages: The model's capability to handle complex multi-source data and enhance prediction accuracy through effective data fusion is advantageous. It captures interactions and dependencies between different sensors, improving overall predictive performance.

In this section, I will compare the characteristics of the Single-Sensor LSTM Encoder-Decoder Model and the Multi-Sensor Fused LSTM Encoder-Decoder Model in terms of data input, information processing, model complexity, and suitable scenarios. The differences between two models are shown in table 4.3.1. The flow chart of the muti-sensor LSTM encoder-decoder short-term prediction model is shown in figure 4.3.4.

Table 4.3.1: Comparison of Single-Sensor and Multi-Sensor LSTM ED Models

Comparison Item	Single-Sensor Model	Multi-Sensor Model
Data Input	Univariate time series	Multivariate time series
Information Processing	Processe each time step individually.	Integrate information from multiple sensors.
Model Complexity	Lower; fewer parameters.	Higher; more parameters due to multivariate input.
Suitable Scenarios	Prediction based on single sensor's data.	Integration of data from multiple sensors.

The flow of the multi-Sensor Enocder-Decoder LSTM short-term prediction model is shown in Algorithm 6.

Algorithm 6 Multi-Sensor Enocder-Decoder LSTM Short-term Prediction

Require: Sensor data ($\{X_i\}$ for each sensor i), window size (w), target length (t), distance threshold ($d_{\text{threshold}}$), correlation threshold ($\rho_{\text{threshold}}$)
Ensure: Predicted sensor data ($\{\hat{X}_i\}$ for each sensor i)

Sensor Network Topology Determination:
Initialize empty directed graph G
for each sensor s_1 in sensor data **do**
    Add node s_1 to G with coordinates $coords(s_1)$
    for each sensor s_2 in sensor data **do**
        if $s_1 \neq s_2$ **then**
            Calculate Euclidean distance d between $coords(s_1)$ and $coords(s_2)$
            Calculate correlation coefficient ρ of $gas\_concentration(s_1)$ and $gas\_concentration(s_2)$
            if $d < d_{\text{threshold}}$ **AND** $|\rho| > \rho_{\text{threshold}}$ **then**
                Add directed edge from s_1 to s_2 in G
            end if
        end if
    end for
end for
Print neighbors of each sensor in G
Multi-Sensor LSTM Encoder-Decoder Short-term Prediction:
Initialize empty list $\{\hat{X}_i\}$
for each sensor i in sensor data **do**
    Normalize and difference X_i
    Initialize encoder-decoder LSTM model
    while Early stopping criterion not met **do**
        for each window in training data of X_i **do**
            $h, c \leftarrow$ Encoder(window data)
            $\hat{X}_{i,\text{window}} \leftarrow$ Decoder(h, c, target length t)
            Update model parameters using backpropagation
        end for
        Evaluate model on validation data
    end while
    Predict next t values using trained model for each window in X_i
    $\hat{X}_i$.append($\hat{X}_{i,\text{window}}$[-1])
    Denormalize $\hat{X}_i$
end for
Evaluation and Visualization:
Visualize original sensor data $\{X_i\}$ and predicted sensor data $\{\hat{X}_i\}$ for each sensor i

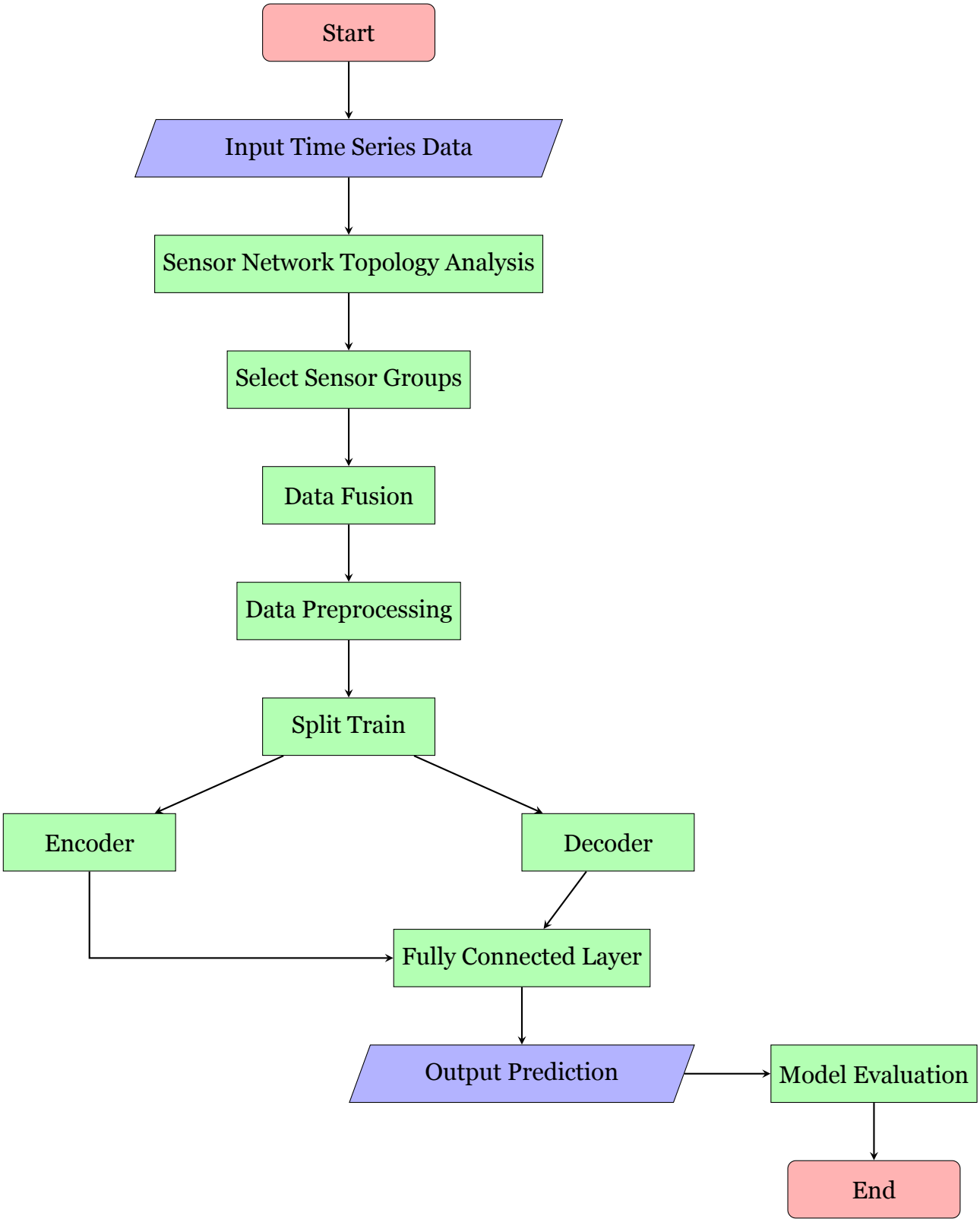


Figure 4.3.4: Flowchart of Multi-sensor LSTM Encoder-Decoder Short-term Prediction

4.4 Gaussian Process Regression Sensor Fusion

4.4.1 Time Dimension Gaussian Process Regression Sensor Fusion

This model predicts gas concentration at a specific coordinate on a map over time, even without a sensor at that location. Imagine you have sensors scattered across a field, measuring gas concentration. This model utilizes readings from these sensors to learn the spatial relationships of gas distribution. By applying Gaussian Process Regression, it estimates the gas concentration at any point on the map, including your target coordinate where no sensor is present. Repeating this process over time allows you to forecast how the gas concentration at that specific point might change, offering valuable insights for monitoring and decision-making.

A Gaussian Process is defined as a collection of random variables, any finite number of which have a joint Gaussian distribution. Formally, a GP can be described by a mean function $m(\mathbf{x})$ and a covariance function $k(\mathbf{x}, \mathbf{x}')$:

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{4.86}$$

The mean function $m(\mathbf{x})$ is often assumed to be zero for simplicity:

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] = 0 \tag{4.87}$$

The covariance function, or kernel $k(\mathbf{x}, \mathbf{x}')$, determines the shape of the functions drawn from the GP.

The Radial Basis Function (RBF) kernel, also known as the Gaussian kernel, is a common choice for $k(\mathbf{x}, \mathbf{x}')$:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2l^2}\right) \tag{4.88}$$

Here, σ_f^2 is the variance parameter and l is the length scale parameter. The length scale l controls the smoothness of the functions.

4.4.1.1 Prediction using GPR

Given training data $\mathbf{X}_{\text{train}}$ with corresponding outputs $\mathbf{y}_{\text{train}}$, I aim to predict the function values $\mathbf{f}_*$ at new input points $\mathbf{X}_*$ using Gaussian Process Regression (GPR). GPR is a non-parametric, Bayesian approach to regression that provides a flexible and probabilistic framework for modeling the relationship between inputs and outputs.

To begin with, I need to construct several covariance matrices that will be used in our predictions. These matrices represent the relationships between different sets of input points.

1. **Covariance matrix of the training points**: This matrix, $K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}})$, encapsulates the pairwise similarities between all training points. Each element of the matrix is computed using a covariance function (kernel), which measures the similarity

between two input points $\mathbf{x}_i$ and $\mathbf{x}_j$. A common choice for the kernel is the squared exponential (or Gaussian) kernel:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-\frac{1}{2}(\mathbf{x}_i - \mathbf{x}_j)^\top \mathbf{L}^{-1}(\mathbf{x}_i - \mathbf{x}_j)\right) \tag{4.89}$$

where σ_f^2 is the signal variance and $\mathbf{L}$ is a diagonal matrix of length scales.

The covariance matrix for the training points is then:

$$K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) = \begin{bmatrix} k(\mathbf{x}_{\text{train},1}, \mathbf{x}_{\text{train},1}) & \cdots & k(\mathbf{x}_{\text{train},1}, \mathbf{x}_{\text{train},n}) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_{\text{train},n}, \mathbf{x}_{\text{train},1}) & \cdots & k(\mathbf{x}_{\text{train},n}, \mathbf{x}_{\text{train},n}) \end{bmatrix} \tag{4.90}$$

2. **Cross-covariance matrix between training and test points**: Next, I will compute the cross-covariance matrix, $K(\mathbf{X}_{\text{train}}, \mathbf{X}_*)$, which represents the covariances between each pair of training and test points. Each element is computed using the same kernel function:

$$K(\mathbf{X}_{\text{train}}, \mathbf{X}_*) = \begin{bmatrix} k(\mathbf{x}_{\text{train},1}, \mathbf{x}_*1) & \cdots & k(\mathbf{x}_{\text{train},1}, \mathbf{x}_*m) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_{\text{train},n}, \mathbf{x}_*1) & \cdots & k(\mathbf{x}_{\text{train},n}, \mathbf{x}_*m) \end{bmatrix} \tag{4.91}$$

3. **Covariance matrix of the test points**: Finally, I will calculate the covariance matrix for the test points, $K(\mathbf{X}_*, \mathbf{X}_*)$, which represents the covariances between each pair of test points, computed using the same kernel function:

$$K(\mathbf{X}_*, \mathbf{X}_*) = \begin{bmatrix} k(\mathbf{x}_*1, \mathbf{x}_*1) & \cdots & k(\mathbf{x}_*1, \mathbf{x}_*m) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_*m, \mathbf{x}_*1) & \cdots & k(\mathbf{x}_*m, \mathbf{x}_*m) \end{bmatrix} \tag{4.92}$$

The next step is to establish the joint distribution of the training outputs and the test function values. The key to GPR is the assumption that the training outputs $\mathbf{y}_{\text{train}}$ and the function values at the test points $\mathbf{f}_*$ follow a joint multivariate normal distribution. This joint distribution is given by:

$$\begin{bmatrix} \mathbf{y}_{\text{train}} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{o}, \begin{bmatrix} K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) + \sigma_n^2 I & K(\mathbf{X}_{\text{train}}, \mathbf{X}_*) \\ K(\mathbf{X}_*, \mathbf{X}_{\text{train}}) & K(\mathbf{X}_*, \mathbf{X}_*) \end{bmatrix}\right) \tag{4.93}$$

Here: - $K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) + \sigma_n^2 I$ is the covariance matrix of the training points with added noise variance σ_n^2 on the diagonal. - $K(\mathbf{X}_{\text{train}}, \mathbf{X}_*)$ is the covariance between the training and test points. - $K(\mathbf{X}_*, \mathbf{X}_{\text{train}})$ is the transpose of $K(\mathbf{X}_{\text{train}}, \mathbf{X}_*)$. - $K(\mathbf{X}_*, \mathbf{X}_*)$ is the covariance matrix of the test points.

Using properties of the multivariate normal distribution, I can derive the predictive distribution for the test outputs $\mathbf{f}_*$. The predictive distribution is also a Gaussian distribution, with the following mean and covariance:

1. **Predictive mean**: The predictive mean represents the expected values of the function at the test points. It is given by:

$$\mathbb{E}[\mathbf{f}_*] = K(\mathbf{X}_*, \mathbf{X}_{\text{train}}) \left(K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) + \sigma_n^2 I\right)^{-1} \mathbf{y}_{\text{train}} \tag{4.94}$$

This expression gives the expected value of the test outputs. The term $K(\mathbf{X}_*, \mathbf{X}_{\text{train}})$ represents the covariance between the test and training points. The term $(K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) + \sigma_n^2 I)^{-1}$ is the inverse of the regularized covariance matrix of the training points, which is used to weight the training outputs $\mathbf{y}_{\text{train}}$.

2. **Predictive covariance**: The predictive covariance quantifies the uncertainty in the predictions. It is given by:

$$\text{Cov}(\mathbf{f}_*) = K(\mathbf{X}_*, \mathbf{X}_*) - K(\mathbf{X}_*, \mathbf{X}_{\text{train}}) \left(K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) + \sigma_n^2 I\right)^{-1} K(\mathbf{X}_{\text{train}}, \mathbf{X}_*) \tag{4.95}$$

This covariance matrix quantifies the uncertainty in the predictions. It accounts for the original covariance of the test points and subtracts the part of the covariance that can be explained by the training data. This subtraction ensures that the remaining covariance reflects only the uncertainty in the test points given the training data.

To summarize, here are the detailed steps involved in making predictions using GPR:

1. **Compute Covariance Matrices**:

(1) Calculate the covariance matrix for the training points $K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}})$.

(2) Calculate the cross-covariance matrix between training and test points $K(\mathbf{X}_{\text{train}}, \mathbf{X}_*)$.

(3) Calculate the covariance matrix for the test points $K(\mathbf{X}_*, \mathbf{X}_*)$.

2. **Formulate the Joint Distribution**:

$$\mathcal{N}\left(\mathbf{o}, \begin{bmatrix} K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) + \sigma_n^2 I & K(\mathbf{X}_{\text{train}}, \mathbf{X}_*) \\ K(\mathbf{X}_*, \mathbf{X}_{\text{train}}) & K(\mathbf{X}_*, \mathbf{X}_*) \end{bmatrix}\right) \tag{4.96}$$

3. **Derive Predictive Mean and Covariance**:

(1) Use the joint distribution to derive the predictive mean:

$$\mathbb{E}[\mathbf{f}_*] = K(\mathbf{X}_*, \mathbf{X}_{\text{train}}) \left(K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) + \sigma_n^2 I\right)^{-1} \mathbf{y}_{\text{train}} \tag{4.97}$$

(2) Use the joint distribution to derive the predictive covariance:

$$\text{Cov}(\mathbf{f}_*) = K(\mathbf{X}_*, \mathbf{X}_*) - K(\mathbf{X}_*, \mathbf{X}_{\text{train}}) \left(K(\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{train}}) + \sigma_n^2 I\right)^{-1} K(\mathbf{X}_{\text{train}}, \mathbf{X}_*) \tag{4.98}$$

Gaussian Process Regression (GPR) provides a probabilistic approach to regression that models the distribution over possible functions that fit the training data. The predictions at new input points $\mathbf{X}_*$ are given by a Gaussian distribution with a mean that represents the predicted values and a covariance that represents the uncertainty of these predictions. This makes GPR a powerful tool for regression tasks where it is important to quantify the uncertainty in the predictions.

4.4.1.2 Workflow Summary

1.Training and Validation Split

To ensure that the model is trained and validated on different portions of the data, I will split each set of 12 data points into three distinct sets: training, validation, and test sets. This split helps in assessing the performance of the model on unseen data and prevents overfitting. Here is the detailed breakdown of how we perform the split:

- **Training set $\mathbf{y}_{train}$**: The training set consists of the first 4 data points. These points are used to fit the model parameters. By training on this subset, the model learns the underlying patterns in the data.

$$\mathbf{y}_{train} = \{y_1, y_2, y_3, y_4\} \tag{4.99}$$

  where y_i represents the i-th data point in the sequence.

- **Validation set $\mathbf{y}_{valid}$**: The validation set includes the next 4 data points. This set is used to tune the hyperparameters of the model and to monitor the model's performance during training. The validation set helps in identifying if the model is overfitting or underfitting.

$$\mathbf{y}_{valid} = \{y_5, y_6, y_7, y_8\} \tag{4.100}$$

  The performance on this set is crucial for selecting the best model configuration.

- **Test set $\mathbf{y}_{test}$**: The test set comprises the last 4 data points. This set is kept aside and only used once the model training and hyperparameter tuning are complete. The performance on the test set provides an unbiased evaluation of the final model.

$$\mathbf{y}_{test} = \{y_9, y_{10}, y_{11}, y_{12}\} \tag{4.101}$$

  The test set results reflect the model's generalization capability to new, unseen data.

2. Hyperparameter Optimization

In Gaussian Process Regression (GPR), the choice of hyperparameters significantly affects the model's performance. One crucial hyperparameter in the Radial Basis Function (RBF) kernel, also known as the Gaussian kernel, is the length scale l. The length scale determines how quickly the covariance between points decreases with distance, influencing the smoothness of the predicted function. Our goal is to find the optimal length scale l that minimizes the Mean Squared Error (MSE) on the validation data.

The Mean Squared Error (MSE) is a measure of the average squared difference between the predicted values $\hat{y}_i$ and the actual values y_i. It is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{4.102}$$

where: - n is the number of data points in the validation set. - $\hat{y}_i$ represents the predicted value for the i-th validation data point. - y_i represents the actual value for the i-th validation data point.

A lower MSE indicates a better fit of the model to the validation data.

To optimize the length scale l of the RBF kernel, I will follow these detailed steps:

1. **Define the Range of Length Scales**: First, I will define a range of potential length scales l to explore. This range should cover a broad spectrum to ensure that the optimal value is included.

2. **Iterate Over Length Scales**: For each candidate length scale l, I will perform the following steps:

   1. **Train the Model**: Train the Gaussian Process model using the training data $\mathbf{X}_{\text{train}}$ and $\mathbf{y}_{\text{train}}$ with the current length scale l. This involves computing the covariance matrix using the RBF kernel:

   $$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-\frac{1}{2l^2}\|\mathbf{x}_i - \mathbf{x}_j\|^2\right) \tag{4.103}$$

   where σ_f^2 is the signal variance, which can be fixed or optimized separately.

   2. **Predict on Validation Set**: Use the trained model to predict the outputs for the validation data $\mathbf{X}_{\text{valid}}$. Obtain the predicted values $\hat{\mathbf{y}}_{\text{valid}}$.

   3. **Compute MSE**: Calculate the MSE between the predicted values $\hat{\mathbf{y}}_{\text{valid}}$ and the actual validation values $\mathbf{y}_{\text{valid}}$:

   $$\text{MSE}(l) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_{\text{valid},i} - y_{\text{valid},i})^2 \tag{4.104}$$

3. **Select Optimal Length Scale**: After evaluating the MSE for each length scale l, I will select the length scale that results in the lowest MSE:

$$l_{\text{optimal}} = \arg\min_l \text{MSE}(l) \tag{4.105}$$

To summarize, the hyperparameter optimization process involves iterating over a range of length scales, training the model for each length scale, predicting the validation data, and computing the MSE. The length scale that yields the lowest MSE on the validation data is chosen as the optimal length scale for the RBF kernel. This ensures that the model generalizes well to unseen data, providing accurate and reliable predictions.

3. Training the Final GPR Model

Using the optimal length scale parameter identified during the model selection process, I will re-train the Gaussian Process Regression (GPR) model on the training data. This refined model is then used to predict values at new test points, and both the predictive mean and variance are computed. The detailed process is as follows:

First, I will prepare the data by segmenting it into different groups for training, validation, and testing:

1. **Data Preparation**:

- **Data Segmentation**: The dataset is segmented into multiple groups. Each group consists of training, validation, and test sets:
  - **Training Data**: The first four columns are designated as the training set.
  - **Validation Data**: The subsequent four columns are used for validation.
  - **Test Data**: The next four columns are used for testing, if available.

  This segmentation ensures that the model can be trained, validated, and tested on different portions of the data, enhancing its generalizability.

Next, I will proceed to train the GPR model using the optimal kernel parameter found earlier:

2. **Model Training**:

- **Optimal Kernel Parameter**: Using the previously determined optimal length scale parameter, ℓ_{best}, the GPR model is re-trained on the full training dataset for each group.
- **Kernel Configuration**: The Radial Basis Function (RBF) kernel is configured with the optimal length scale parameter to achieve the best predictive performance. The RBF kernel function is defined as:

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\ell_{\text{best}}^2}\right) \tag{4.106}$$

  where ℓ_{best} is the optimal length scale parameter.
- **Training**: The GPR model is then fitted to the training data $\{(x_i, y_i)\}_{i=1}^{N}$, where x_i are the input features and y_i are the corresponding target values. This fitting process involves optimizing the model parameters to best capture the underlying relationships in the training data.

Finally, I will use the trained model to make predictions at new test points and compute the associated uncertainties:

3. **Prediction at New Test Points**:

- **Test Points Definition**: Define the new test points $\{x_j^*\}_{j=1}^{M}$ where predictions are desired. These test points are chosen based on the specific application and the need for prediction at these locations.
- **Predictive Mean and Variance Calculation**:
  - **Predictive Mean**: The expected value of the target at each test point x^* is computed using:

$$\mu(x^*) = k(x^*)^T [K + \sigma_n^2 I]^{-1} y \tag{4.107}$$

Here, $k(x^*)$ is the covariance vector between the test point x^* and the training points, K is the covariance matrix of the training points, σ_n^2 is the noise variance, and I is the identity matrix.

– **Predictive Variance**: The uncertainty of the prediction at each test point x^* is computed as:

$$\sigma^2(x^*) = k(x^*, x^*) - k(x^*)^T[K + \sigma_n^2 I]^{-1}k(x^*) \tag{4.108}$$

where $k(x^*, x^*)$ is the covariance of the test point with itself.

- **Predictive Results**: The predictive mean $\{\mu(x_j^*)\}_{j=1}^M$ and variance $\{\sigma^2(x_j^*)\}_{j=1}^M$ are generated for the defined test points. The predictive mean provides the estimated target values, while the predictive variance quantifies the uncertainty of these estimates.

By following these steps, the final GPR model is trained with the most optimal configuration, allowing for reliable predictions and accurate uncertainty estimates for new test points. This process ensures that the model is both accurate and robust, making it suitable for practical applications where prediction reliability is crucial.

4. Performance Evaluation

To comprehensively assess the effectiveness of the Gaussian Process Regression (GPR) model, I will utilize the Mean Absolute Error (MAE) metric. The MAE quantifies the average magnitude of errors between predicted and actual data points, providing a clear measure of the model's predictive accuracy. Here's a detailed breakdown of our evaluation process:

Firstly, I will begin by generating predictions using the trained GPR model on our test dataset:

1. **Model Predictions**:

- Following the training phase with optimized kernel parameters, the GPR model is applied to predict outcomes on the test dataset.
- Let $\hat{y}_i$ represent the predicted value for the i-th test instance, and y_i denote its corresponding actual value.

Subsequently, I will calculate the MAE, which captures the average absolute discrepancy between predicted and actual values:

2. **Mean Absolute Error Calculation**:

- The MAE is computed by averaging the absolute differences across all test instances:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \tag{4.109}$$

where n denotes the total number of test points, $\hat{y}_i$ signifies the predicted value, and y_i denotes the actual value.

Lastly, I will summarize the evaluation process to provide a comprehensive assessment:

3. **Evaluation Process Overview**:

- Post-prediction, absolute errors are computed for each test point to quantify prediction accuracy.
- These individual errors are then aggregated and averaged to derive the MAE, offering a consolidated metric of overall model performance.

By adhering to these steps, I will ensure a thorough evaluation of the GPR model's efficacy in predicting new data points. The MAE metric serves as a pivotal benchmark, facilitating comparisons across different models or parameter configurations.

The pseudo-code for the model implementation step is shown in Algorithm 7. The flowchart of the model is shown in figure 4.4.1.

Algorithm 7 GPR Time Dimension Prediction

Require:
Time series: $\{y_t\}_{t=1}^{T}$
Input features: $\mathbf{X} = \{\mathbf{x}_t\}_{t=1}^{T}$
Window size: w
Kernel function set: $\mathcal{K}$
Ensure:
Predicted values: $\{\hat{y}_t\}_{t=w+1}^{T}$
procedure GPR-Predict($y_{1:T}$, $\mathbf{X}$, w, $\mathcal{K}$)
    for $t = w + 1$ **to** T **do**
        Training data: $\mathbf{Y}_{train} \leftarrow (y_{t-w}, y_{t-w+1}, ..., y_{t-1})$
        Training inputs: $\mathbf{X}_{train} \leftarrow (\mathbf{x}_{t-w}, \mathbf{x}_{t-w+1}, ..., \mathbf{x}_{t-1})$
        $best\_score \leftarrow -\infty$
        $best\_model \leftarrow null$
        for $k \in \mathcal{K}$ **do**
            Train GPR model gp with kernel k on $(\mathbf{X}_{train}, \mathbf{Y}_{train})$
            $score \leftarrow Evaluate(gp, (\mathbf{X}_{train}, \mathbf{Y}_{train}))$
            if $score > best\_score$ **then**
                $best\_score \leftarrow score$
                $best\_model \leftarrow gp$
            end if
        end for
        $\hat{y}_t \leftarrow best\_model.predict(\mathbf{x}_t)$
    end for
    return $\{\hat{y}_t\}_{t=w+1}^{T}$
end procedure

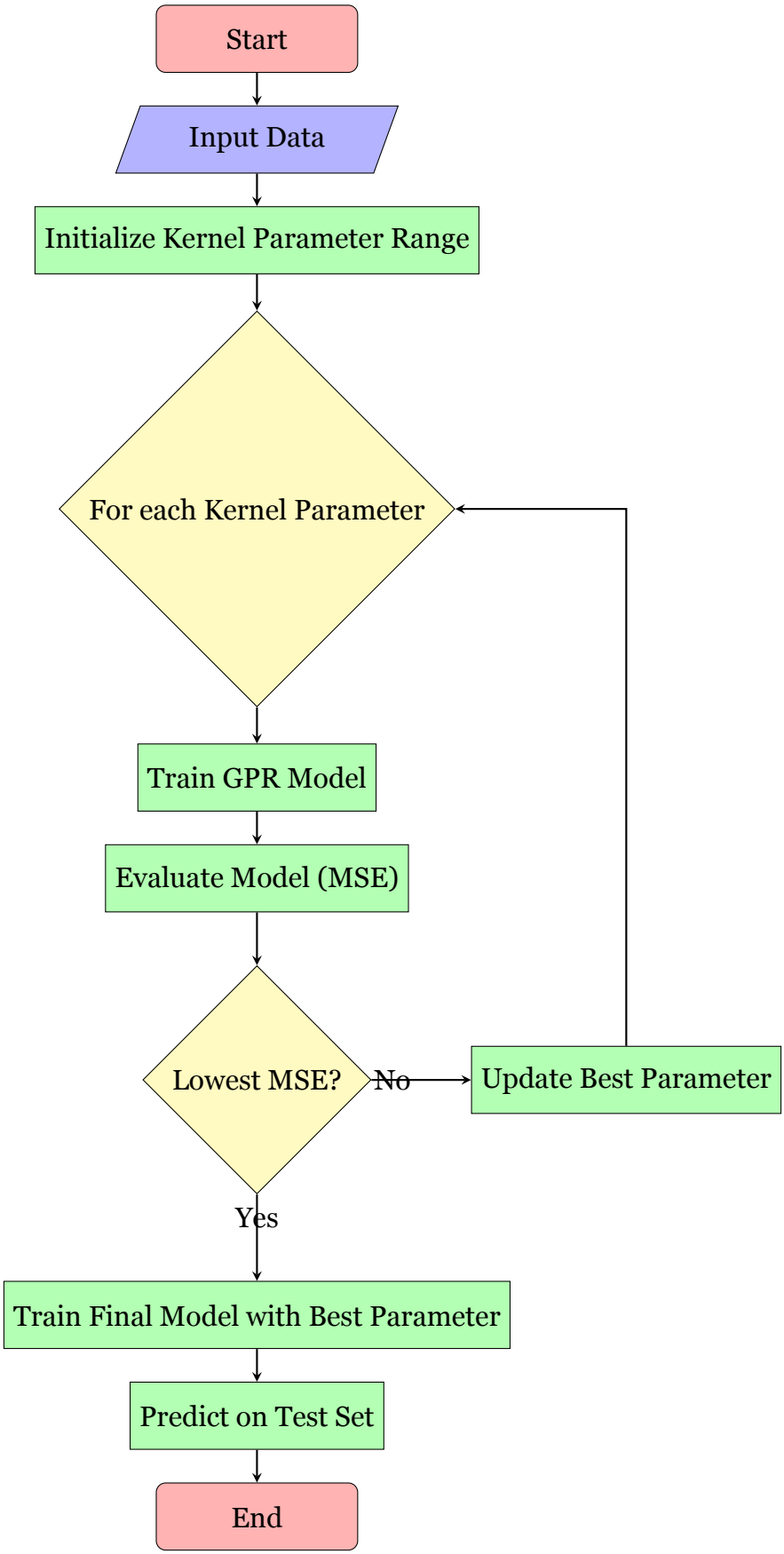


Figure 4.4.1: Flowchart of GPR for Time Dimension Prediction

4.4.2 Space Dimension Gaussian Process Regression Sensor Fusion

This model aims to predict the complete gas concentration distribution across an entire map at a specific moment in time. Imagine a snapshot of gas levels across a region. Using readings from strategically placed sensors, the model leverages Gaussian Process Regression to interpolate and extrapolate these measurements. This creates a continuous surface, effectively representing the estimated gas concentration at every point on the map, even in areas without sensor coverage. This provides a comprehensive view of gas distribution at that particular time point, enabling better understanding of spatial patterns and potential hotspots.

The Gaussian process regression sensor fusion prediction in the spatial dimension is similar to the Gaussian process regression sensor fusion prediction process in the time dimension. The only difference is that the training data and test data set are changed. Therefore, I will not introduce the model in the spatial dimension in detail here, but only compare and analyze the differences between the two models.

1. Data Representation and Handling:

(a) Time Dimension Gaussian Process Regression Sensor Fusion

In this application, GPR is employed to fuse temporal data from multiple sensors, treating time as the primary dimension:

- **Introduction**: In temporal sensor fusion, data from multiple sensors are collected over time. Gaussian Process Regression (GPR) is employed to model and fuse these temporal data streams.
- **Data Representation**: Each sensor provides sequential readings over time. Let $Y_i(t)$ denote the reading from sensor i at time t, where $i = 1, 2, 3, 4$.
- **Gaussian Process Model**: The temporal process $Y(t)$ is modeled as a Gaussian Process with mean function $m(t)$ and covariance function $k(t, t')$:

$$Y(t) \sim \mathcal{GP}(m(t), k(t, t')) \tag{4.110}$$

  Commonly used covariance functions include the squared exponential kernel:

$$k(t, t') = \sigma^2 \exp\left(-\frac{(t - t')^2}{2\ell^2}\right) \tag{4.111}$$

  where σ^2 controls the variance and ℓ controls the lengthscale of temporal correlations.

- **Objective**: The objective of using GPR in this context is to predict $Y(t)$ and estimate uncertainty over time, leveraging data from all sensors to improve prediction accuracy.

(b) Space Dimension Gaussian Process Regression Sensor Fusion

In spatial sensor fusion, Gaussian Process Regression (GPR) is applied to integrate measurements from sensors arranged in a 2D grid:

- **Introduction**: Spatial sensor fusion involves aggregating data from sensors positioned across a 2D grid. GPR serves as a powerful tool to model spatial correlations and create unified spatial maps.
- **Data Representation**: Sensors provide measurements at different grid coordinates. Let $Y(x, y)$ denote the measured value at coordinates (x, y).
- **Gaussian Process Model**: The spatial process $Y(x, y)$ is modeled as a Gaussian Process with mean function $m(x, y)$ and covariance function $k((x, y), (x', y'))$:

$$Y(x, y) \sim \mathcal{GP}(m(x, y), k((x, y), (x', y'))) \tag{4.112}$$

  The covariance function often uses a squared exponential kernel:

$$k((x, y), (x', y')) = \sigma^2 \exp\left(-\frac{(x - x')^2 + (y - y')^2}{2\ell^2}\right) \tag{4.113}$$

  where σ^2 controls the variance and ℓ controls the lengthscale of spatial correlations.
- **Objective**: Gaussian process regression helps create a smooth spatial map $Y(x, y)$ of CO_2 concentrations, integrating data from all sensors to provide a comprehensive view of the spatial distribution.

2. Kernel Optimization and Parameter Tuning

(a) Time Dimension Gaussian Process Regression Sensor Fusion

In this method, we aim to optimize a 2D Radial Basis Function (RBF) kernel independently for each temporal group of sensor readings. The goal is to iteratively search for the optimal length scale parameter that minimizes Mean Squared Error (MSE) during the training process.

The RBF kernel k_{time} for the time dimension can be defined as:

$$k_{\text{time}}(t_i, t_j) = \sigma_{\text{time}}^2 \exp\left(-\frac{(t_i - t_j)^2}{2l_{\text{time}}^2}\right) \tag{4.114}$$

where t_i and t_j are timestamps of sensor readings, σ_{time}^2 is the variance parameter, and l_{time} is the length scale parameter specific to the time dimension.

During training, we iteratively adjust l_{time} to minimize MSE, which effectively adapts the kernel's sensitivity to temporal variations in the sensor data.

(b) Space Dimension Gaussian Process Regression Sensor Fusion

Unlike the temporal optimization, this approach involves optimizing a 2D RBF kernel across a spatial grid. Here, we systematically explore combinations of length scale parameters $l_{\text{space},x}$ and $l_{\text{space},y}$ for the X and Y dimensions, respectively, to minimize MSE across the entire spatial domain.

The spatial RBF kernel k_{space} is defined as:

$$k_{\text{space}}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_{\text{space}}^2 \exp\left(-\frac{(\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{L}_{\text{space}} (\mathbf{x}_i - \mathbf{x}_j)}{2}\right) \tag{4.115}$$

where $\mathbf{x}_i$ and $\mathbf{x}_j$ are spatial coordinates, σ^2_{space} is the variance parameter, and $\mathbf{L}_{\text{space}} = \text{diag}(l^2_{\text{space},x}, l^2_{\text{space},y})$ is a diagonal matrix of length scale parameters for X and Y dimensions.

By exploring various combinations of $l_{\text{space},x}$ and $l_{\text{space},y}$, we aim to find the optimal spatial scale that best captures the spatial relationships and variations present in the sensor data.

3. Prediction and Evaluation Metrics

(a) Time Dimension Gaussian Process Regression Sensor Fusion

This aspect of the model predicts CO_2 concentrations at a specific fixed point ($X_{\text{test}} = [20, 24]$) after being trained on historical sensor data. The accuracy of predictions is assessed using key metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE). These metrics quantify how closely the predicted values align with actual sensor readings, providing insights into the model's performance in capturing temporal variations in CO_2 levels.

(b) Space Dimension Gaussian Process Regression Sensor Fusion

Here, the model predicts CO_2 concentrations across a dense 2D grid, defined by coordinates generated using np.meshgrid. By leveraging Gaussian Process Regression, the model estimates CO_2 levels spatially, offering predictions for every point within the grid. Evaluation involves visual inspection through 3D surface plots and contour maps. These visual representations illustrate the spatial distribution of predicted CO_2 concentrations, allowing for qualitative assessment of how well the model captures variations across different spatial coordinates.

5. Application Context and Suitability:

(a) Time Dimension Gaussian Process Regression Sensor Fusion

Suitable for applications requiring real-time or sequential sensor data fusion, such as predictive maintenance or time-series analysis. Gaussian Process Regression (GPR) excels in scenarios where data points are collected over time and where the relationships between data points are non-linear or complex. By leveraging GPR, systems can predict future sensor readings with uncertainty estimates, enabling proactive maintenance schedules or predictive analytics to optimize operations in dynamic environments.

(b) Space Dimension Gaussian Process Regression Sensor Fusion

Ideal for applications involving spatial mapping and environmental monitoring where understanding spatial distribution and variability of CO_2 concentrations is critical. In spatial contexts, GPR integrates multiple sensor inputs to create a detailed spatial map of CO_2 levels, identifying hotspots or areas of interest with high precision. This capability is invaluable for environmental monitoring, urban planning, or indoor air quality management, where accurate spatial insights aid decision-making and intervention strategies.

Chapter 5

Result

5.1 Sensor Allocation and Data Collection

The sensors were initially calibrated on November 21 at Malvinas väg 10 . The following morning, they were placed in the Digital Futures building, whose map is shown in figure 5.1.1. Sensor 00 and Sensor 01 were both located in the central area; therefore, only the data from Sensor 01 were considered for analysis to avoid redundancy. Sensor 02 was positioned on the left side, monitoring that specific area, while Sensor 03 was placed in the upper right section, likely a conference room. Sensor 04 was located in the right central area, covering another meeting space.

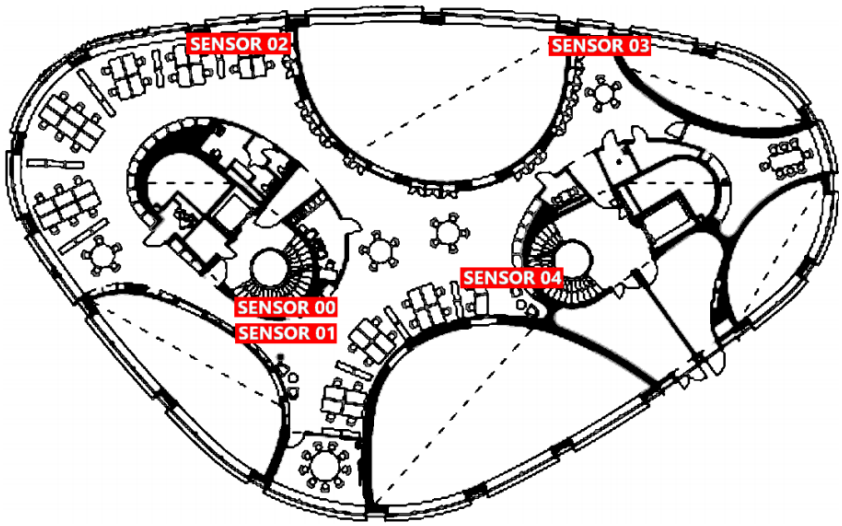


Figure 5.1.1: Map of KTH Digital Future

The collected data showed CO_2 concentrations increasing during the daytime, correlating with occupant activity, and decreasing at night, indicating effective ventilation. Comparisons of different sensors revealed that the LP8 sensor's data were uncalibrated and less stable, consistent with its older model status. This setup provided comprehensive air quality monitoring throughout the building. Figure 5.1.2 shows the raw data of the low-cost sensor lp8. Figure 5.1.3 shows the sensor data comparison between the low-cost sensor lp8 03 and the high-cost sensor sunrise 03.

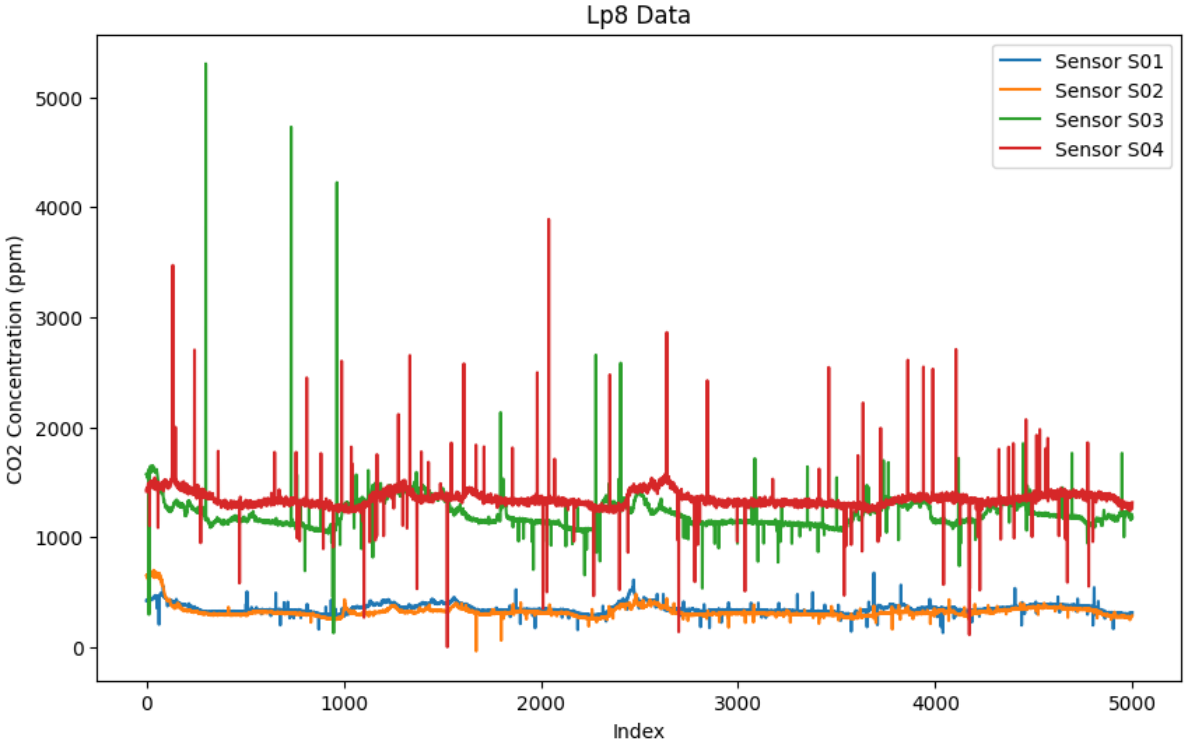


Figure 5.1.2: Original Data of Lp8 Sensors

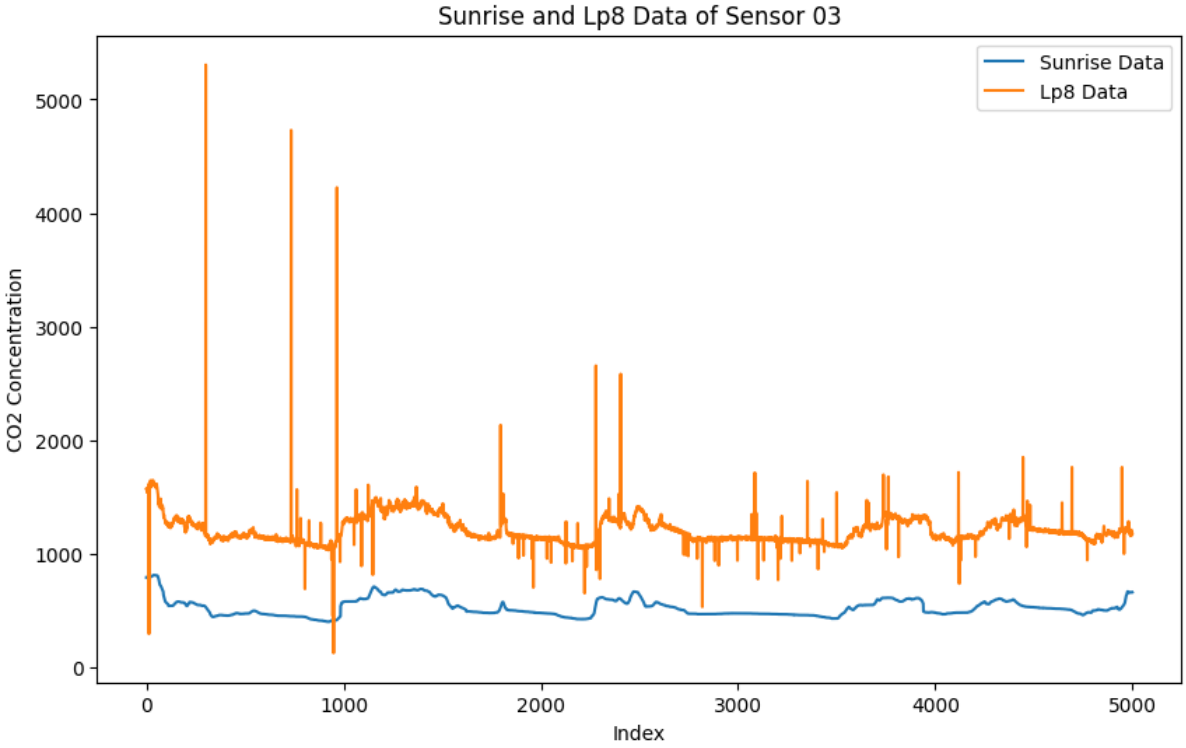


Figure 5.1.3: Original Data of Lp8 Sensors

The raw readings from the LP8 sensor exhibit numerous outliers in the form of protrusions and depressions. These irregularities pose significant challenges for subsequent data processing tasks. Outliers can skew statistical analyses, distort trend identification, and undermine the reliability of the data. In contrast, when compared to high-precision sensors, noticeable discrepancies are evident not only in accuracy but also in the smoothness of the data curve, which is directly impacted by the presence of outliers. The erratic nature of the Lp8 sensor's output introduces noise that complicates data interpretation and algorithmic processing. Addressing these outliers is crucial to enhance the sensor's performance and ensure the integrity of subsequent analytical processes. Techniques such as outlier detection algorithms or filtering methods may be employed to mitigate these irregularities and improve the overall quality of the sensor data for reliable decision-making and analysis.

5.2 ARIMA Outlier Detection

5.2.1 Parameter Estimation

ARIMA (AutoRegressive Integrated Moving Average) models are widely used for time series forecasting. These models are denoted as ARIMA(p, d, q), where: p is the number of autoregressive terms, d is the number of nonseasonal differences needed for stationarity, q is the number of lagged forecast errors in the prediction equation.

In order to determine the three parameters of the model, two steps are required. First, the original data is processed by first-order difference to determine whether the sequence is stable after the first-order difference. If it is stable after the first-order difference, the number of differences d is 1. Next, the acf and pacf of the original data are plotted. By observing the acf and pacf values in these two figures, the other two parameters p and q can be determined.

First, check if the original data is stationary. Typically, methods like the ADF test are used. Suppose we have decided to difference the data once (d=1) to make it stationary.

In the PACF plot (figure 5.2.1(b)), observe the significant lags that exceed the confidence interval. The PACF plot shows the partial correlation of the time series with its own lagged values. The first lag shows a significant negative partial autocorrelation, indicating a strong autoregressive relationship at this lag. Subsequent lags gradually decrease and approach zero, indicating that further lags do not have significant partial autocorrelations.

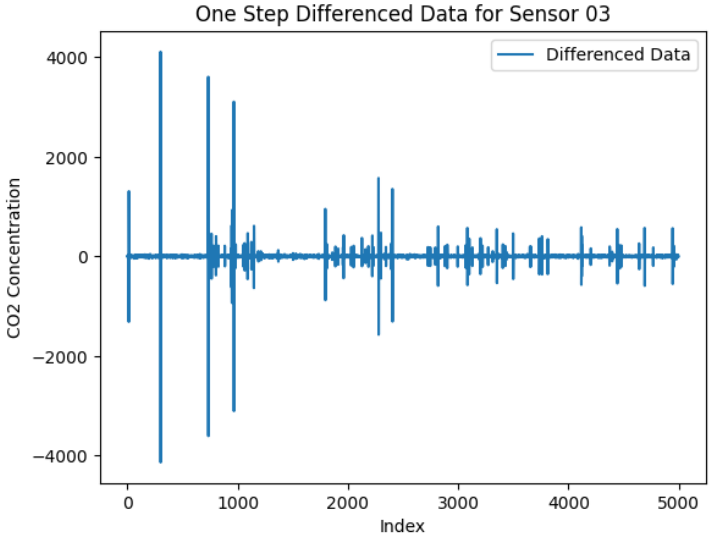
Based on the PACF plot, select the first significant lag as the order of the AR model. Hence, choose $p = 1$.

In the ACF plot (figure 5.2.1(b)), observe the significant lags that exceed the confidence interval. The ACF plot shows the correlation of the time series with its own lagged values. The first lag shows a significant negative autocorrelation, indicating a strong moving average relationship at this lag. Subsequent autocorrelations are close to zero, suggesting that most of the autocorrelation in the differenced data has been

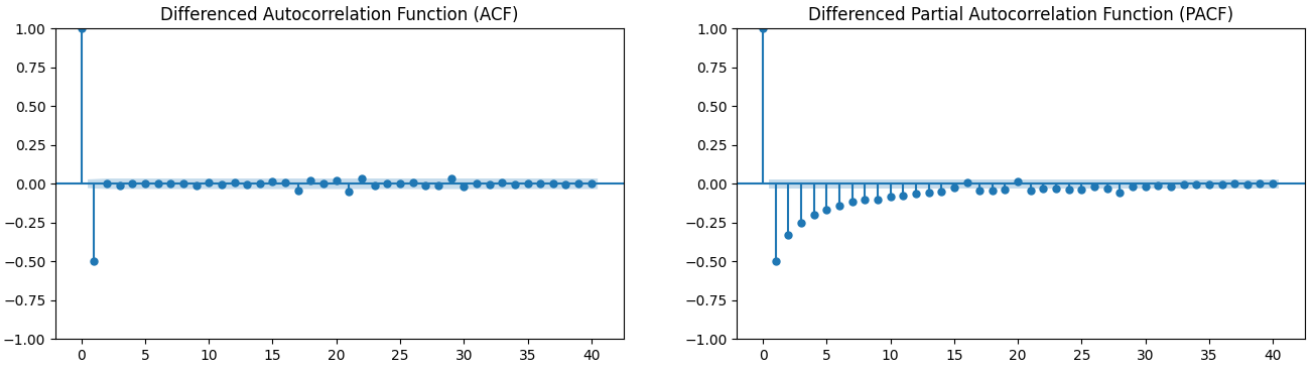
removed.

Based on the ACF plot, select the first significant lag as the order of the MA model. Hence, choose $q = 1$.

Thus, the suggested ARIMA model is ARIMA(1, 1, 1). This model should be a good starting point for fitting the time series data.



(a) First-difference Data



(b) ACF and PACF of Data

Figure 5.2.1: Figure to estimate parameter

5.2.2 ARIMA Outlier Detection and Processing

This section explores the methods used for outlier detection and the approach taken for outlier handling.

To identify outliers, the Z-score method is employed due to its statistical effectiveness in measuring the deviation of individual data points from the mean. The Z-score, Z_i, for a given residual e_i is calculated using the formula:

$$Z_i = \frac{e_i - \bar{e}}{\sigma_e} \tag{5.1}$$

where:

- e_i is the residual at time i,
- $\bar{e}$ is the mean of the residuals,
- σ_e is the standard deviation of the residuals.

Outlier detection involves several systematic steps to ensure accurate identification and handling of outliers:

1. **Model Fitting**: Before outlier detection, an ARIMA model must be fitted to the time series data. This model captures the underlying trend, seasonality, and autocorrelation structure.
2. **Residual Calculation**: After fitting the model, residuals are computed as the differences between the observed values and the model's predictions. These residuals reveal the discrepancies that the model fails to explain:

$$e_i = \textit{Actual\_Value} - \textit{Predicted\_Value} \tag{5.2}$$

3. **Compute Mean and Standard Deviation of Residuals**: Statistical metrics such as the mean ($\bar{e}$) and standard deviation (σ_e) of the residuals are then computed. These metrics provide a basis for standardizing the residuals:

$$\bar{e} = \frac{1}{n}\sum_{i=1}^{n} e_i \tag{5.3}$$

$$\sigma_e = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(e_i - \bar{e})^2} \tag{5.4}$$

4. **Z-score Calculation**: Each residual is standardized using the Z-score equation 5.1. The Z-score transforms residuals into a standard scale, indicating the number of standard deviations each residual deviates from the mean.
5. **Identify Outliers**: Outliers are identified by applying a threshold to the Z-scores. A threshold of $|Z_i| > 1.5$ is commonly used, where any residual with a Z-score greater than 1.5 or less than -1.5 is flagged as an outlier. This threshold is chosen to capture deviations beyond typical fluctuations.

After identifying outliers using Z-score, the next step involves handling them to ensure smooth and continuous data for modeling. Linear interpolation is chosen as the method for handling outliers, which involves estimating an interpolated value $\hat{y}_{\text{interp}}$ for the outlier y_t using neighboring data points y_{t-1} and y_{t+1}:

$$\hat{y}_{\text{interp}} = y_{t-1} + \frac{(y_{t+1} - y_{t-1})}{2} \tag{5.5}$$

Here's a breakdown of the components involved:

- y_{t-1}: The value of the time series data point immediately before the outlier y_t.

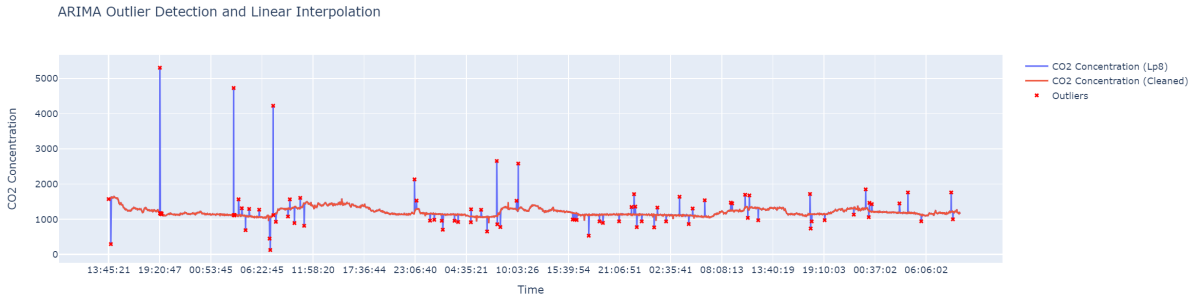


Figure 5.2.2: Outliers Detection and Linear Interpolation

- y_{t+1}: The value of the time series data point immediately after the outlier y_t.
- $\hat{y}_{\text{interp}}$: The interpolated value of the outlier y_t, calculated as the average of its neighboring data points.

This linear interpolation formula calculates $\hat{y}_{\text{interp}}$ by taking the average of y_{t-1} and y_{t+1}. This approach effectively smooths out the abrupt deviation caused by the outlier while preserving the overall trend of the time series. Linear interpolation is particularly suitable when the time series exhibits gradual changes between adjacent data points.

Figure 5.2.2 illustrates the results of outlier detection using the ARIMA model. The blue line represents the original CO_2 concentration data, while the red line shows the cleaned data after replacing outliers using linear interpolation. The red crosses indicate the detected outliers. Evidently, the ARIMA model successfully identified several outliers that deviate significantly from the general trend of CO_2 concentration. By replacing these outliers, the cleaned data provides a more accurate representation of CO_2 concentration over time.

After replacing the outlier with $\hat{y}_{\text{interp}}$, the time series is ready for further analysis or modeling. This method ensures that the data remains coherent and minimizes the disruptive effects of outliers on subsequent statistical modeling techniques like ARIMA.

By integrating linear interpolation into the outlier handling process, the ARIMA model can more accurately capture the underlying patterns and provide reliable forecasts based on the adjusted data. This approach balances simplicity with effectiveness in maintaining the integrity of the time series data.

Figure 5.2.3 presents sunrise sensor data and processed Lp8 data. The blue line representing sunrise data shows a regular pattern with fluctuations within a consistent range, typically between approximately 400 and 700. This pattern reflects the predictable daily cycle of sunrise, showcasing its stability over time.

In contrast, the orange line representing the processed Lp8 data exhibits a broader range, typically fluctuating between 1100 and 1500. This dataset displays more pronounced fluctuations compared to the sunrise data, indicating variability in CO_2 concentration levels over time.

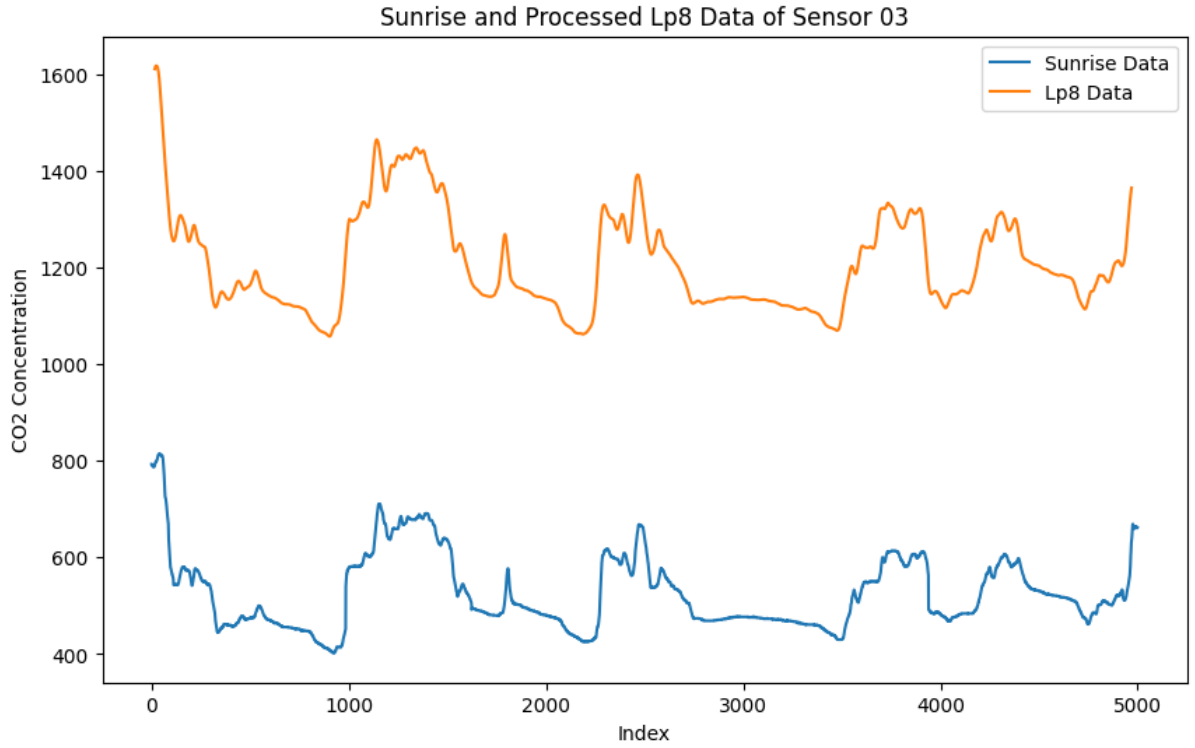


Figure 5.2.3: Processed Data of Lp8 Sensor

A key observation from the graph is the effective impact of the outlier removal process on the processed Lp8 data. While there is no direct overlay or evident correlation between the sunrise and processed Lp8 data, the graph demonstrates a significant improvement in data quality post-outlier removal. The processed Lp8 data now displays a smoother and more consistent trend compared to its raw form, suggesting successful elimination of spurious spikes or dips that may have initially obscured the underlying CO_2 concentration pattern.

This enhanced data presentation underscores the importance of outlier detection and removal in improving the reliability and interpretability of time-series data analyses, particularly in environmental monitoring contexts. By removing outliers, the processed Lp8 data offers a clearer and more accurate representation of CO_2 concentration trends, aiding in more informed decision-making and scientific understanding.

Overall, the graph effectively illustrates the benefits of rigorous data preprocessing techniques, highlighting how outlier removal can enhance the clarity and utility of environmental data visualizations and analyses.

The effectiveness of outlier handling in ARIMA modeling can be visually demonstrated through data presentation and analysis. Outliers, which are extreme values deviating significantly from the overall pattern of the data, can distort model accuracy and prediction reliability. In the context of ARIMA, identifying and appropriately handling outliers is crucial to improve model performance.

One effective method to evaluate the impact of outlier handling is through the

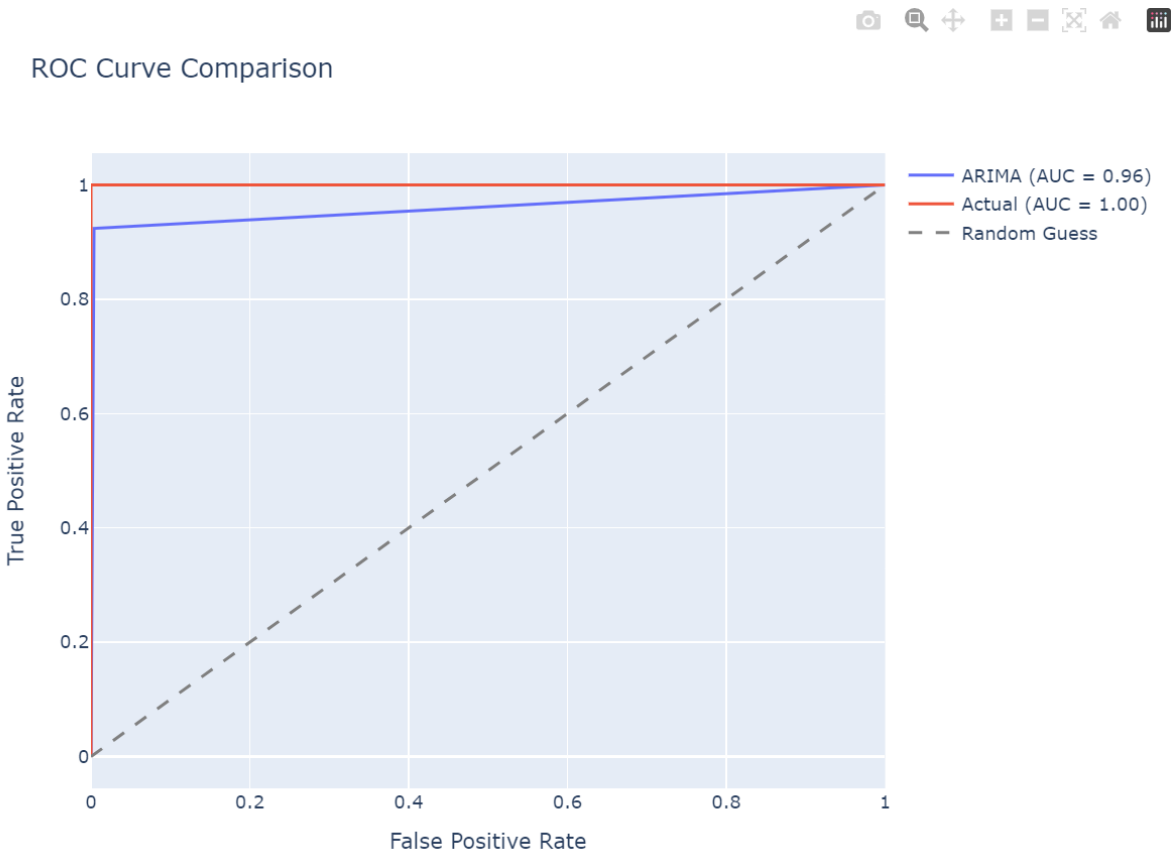


Figure 5.2.4: ARIMA Outlier Detection ROC

Receiver Operating Characteristic (ROC) curve. This statistical tool measures the trade-off between true positive rate (sensitivity) and false positive rate (1 - specificity). By comparing ROC curves before and after outlier handling, we can quantify the improvement in model performance.

Mathematically, the ROC curve is constructed by plotting the true positive rate (sensitivity) against the false positive rate (1 - specificity) at various threshold settings. The true positive rate (TPR) is defined as:

$$TPR = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{5.6}$$

And the false positive rate (FPR) is defined as:

$$FPR = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} \tag{5.7}$$

The area under the ROC curve (AUC) quantifies the overall performance of the model. A higher AUC indicates better discrimination between true positives and false positives, demonstrating improved model accuracy after outlier handling.

Figure 5.2.4 shows that the ARIMA model has excellent performance (AUC = 0.96) in a binary classification task, closely approaching the ideal performance of using the true labels (AUC = 1.00) and significantly outperforming a random classifier.

5.3 DBCAN Outlier Detection

Here, for comparison, I will use the dbscan method to handle outliers. However, this method is not used as an outlier handling method in the final result, so I will only show the processing results without going into too much detail.

Figure 5.3.1 illustrates the application of DBSCAN, a clustering algorithm used for outlier detection, followed by linear interpolation to manage outliers in CO_2 value data over time.

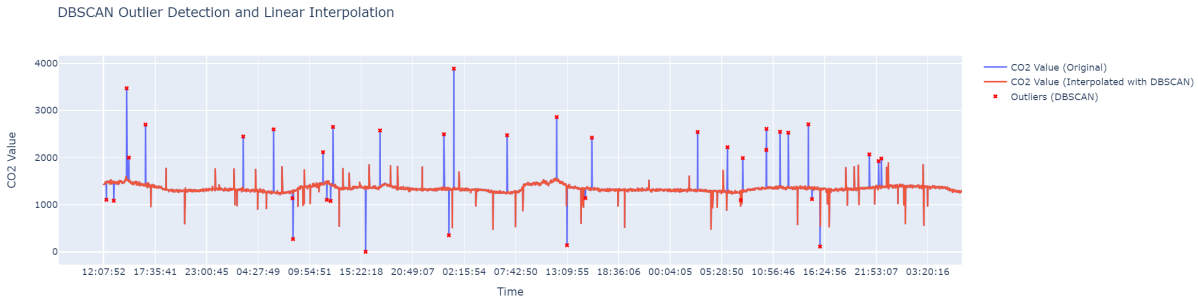


Figure 5.3.1: DBSACN Outlier Detection

To begin with, the figure 5.3.1 includes:

- **Original CO_2 Values (Blue Line):** Shows the raw data points of CO_2 values over time, including spikes that deviate from the general trend.
- **Processed Data (Red Line):** Represents CO_2 values after applying DBSCAN for outlier detection and subsequent linear interpolation. The red line shows a smoother curve compared to the original data due to outlier removal and interpolation.
- **Outliers (Red Crosses):** Scattered red crosses highlight outliers identified by DBSCAN. These outliers typically correspond to extreme spikes or values that do not conform to the overall trend.

Moving forward, I will explaine of DBSCAN and interpolation:

- **DBSCAN:** Used to detect outliers based on data point density in the CO_2 value time series. Points in low-density areas or not well-clustered are flagged as outliers (marked by red crosses).
- **Linear Interpolation:** After identifying and removing outliers with DBSCAN, linear interpolation fills gaps left by these outliers. It estimates values for missing data points by drawing straight lines between remaining data points.

Moreover, consider the impact and benefits of this approach:

- **Data Smoothing:** Linear interpolation smooths CO_2 data by filling gaps from outlier removal, resulting in a visually smoother trend (represented by the red line).

- **Improved Reliability:** Mitigating extreme values (outliers) enhances the reliability of the processed data (red line) in reflecting CO_2 levels over time more accurately, crucial for analysis and decision-making.

Figure 5.3.1 effectively demonstrates using DBSCAN for outlier detection and subsequent linear interpolation to handle outliers in CO_2 value time series data. This approach improves data quality by smoothing trends and ensuring more reliable representation of CO_2 levels over time.

In figure 5.3.2, the ROC curve comparison among DBSCAN, Actual, and Random Guess models offers a clear depiction of their classification performances based on Area Under the Curve (AUC) scores. The Actual model stands out prominently with an AUC of 1.00, indicating flawless classification ability. Its ROC curve forms a perfect right angle at the top left corner, demonstrating ideal separation between positive and negative samples.

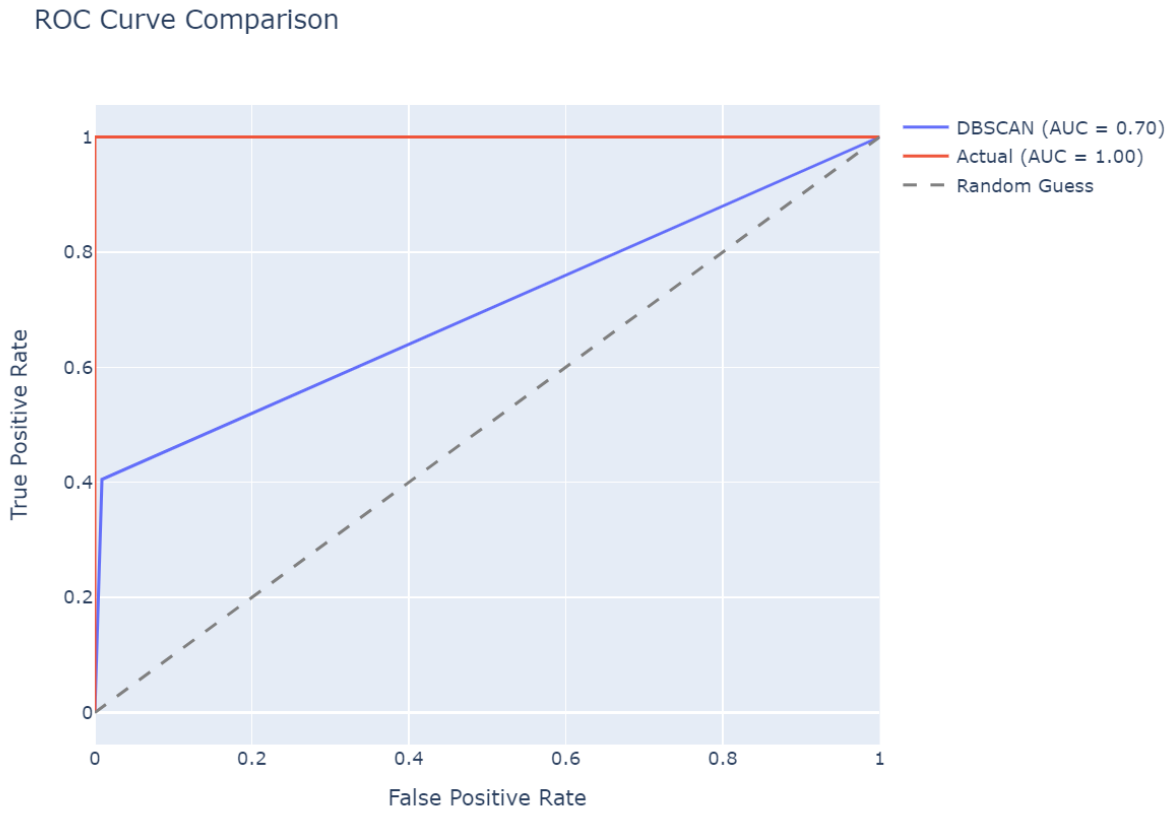


Figure 5.3.2: DBSACN Outlier Detection ROC

In contrast, the DBSCAN model, with an AUC of 0.70, shows decent performance. While it surpasses random guessing (AUC = 0.50), it does not achieve the perfection seen in the Actual model. The DBSCAN curve deviates noticeably from the diagonal line, showcasing its ability to distinguish between classes better than random chance but with some misclassifications.

The dashed diagonal line represents the Random Guess baseline, which implies no

predictive power (AUC = 0.50). Models with AUCs below this line are worse than random guessing, while those above it demonstrate some predictive capability.

While the DBSCAN model exhibits promising discriminatory power compared to randomness, there remains room for improvement to approach the perfect classification achieved by the Actual model. Enhancing the model's performance could involve refining parameters, feature selection, or considering alternative algorithms to achieve higher accuracy and reliability in classification tasks.

In comparing ARIMA and DBSCAN for outlier detection, several key factors contribute to understanding why ARIMA may outperform DBSCAN:

1. **Data Characteristics and Distribution**:
    - **DBSCAN** is effective for datasets with distinguishable density-based separations. It relies on the density of neighboring points to classify outliers. If outliers do not exhibit clear separation in density from normal points, DBSCAN may fail to detect them accurately.
    - **ARIMA**, on the other hand, is typically used for time series data, capable of capturing temporal correlations and trends. If outliers manifest as significant deviations in time series (e.g., unusually high or low values), ARIMA can detect these anomalies by modeling and forecasting residuals.
2. **Types of Outliers**:
    - **DBSCAN** is primarily suited for density-based outlier detection, identifying isolated points or low-density regions in a dataset. Outliers that do not conform to density-based separation criteria may be missed by DBSCAN.
    - **ARIMA** is better suited for detecting impactful anomalies within time series data, such as sudden spikes or prolonged deviations from expected trends.
3. **Model Complexity and Parameter Tuning**:
    - **DBSCAN**'s
      performance depends heavily on parameters like `eps` (neighborhood size) and `min_samples` (minimum number of samples for a core point). Proper parameter selection is critical for effective outlier detection.
    - **ARIMA** requires selecting appropriate orders (`p, d, q`) that reflect the autoregressive, differencing, and moving average components of the data. Correct parameterization enhances ARIMA's ability to capture outliers accurately.
4. **Data Volume and Dimensionality**:
    - **DBSCAN** may struggle with high-dimensional data due to increased complexity in distance calculations and density estimation. Outliers dispersed across multiple dimensions may evade detection by DBSCAN.
    - **ARIMA**, however, is not directly applicable to high-dimensional data as it focuses on modeling and forecasting time-dependent variations.

In conclusion, ARIMA's superiority in outlier detection over DBSCAN can be attributed to its ability to leverage temporal dynamics and residual analysis in time series data. DBSCAN excels in density-based outlier detection scenarios with lower-dimensional data exhibiting clear density separations. Choosing the appropriate method depends on understanding the specific data characteristics and outlier detection requirements of the problem at hand.

5.4 DeepCM Sensor Calibration

5.4.1 Supervised DeepCM Sensor Calibration

Figure 5.4.1 illustrates the supervised calibration process for Sensor 02, providing a visual representation of the raw and calibrated CO_2 concentration measurements. This figure includes three distinct lines: the blue line representing the Lp8 Data, the orange line representing the Sunrise Data, and the green line representing the Calibrated Data.

The blue line, labeled as Lp8 Data, displays the raw CO_2 concentration measurements obtained from the Lp8 sensor. These measurements show significant deviations and fluctuations, underscoring the inherent inaccuracies and inconsistencies in the raw sensor data. The erratic nature of the blue line highlights the necessity for a calibration process to ensure more reliable and accurate CO_2 readings.

The orange line, labeled as Sunrise Data, serves as the reference or ground truth for the calibration process. This line represents CO_2 concentration measurements from a more reliable and stable reference source. Compared to the Lp8 Data, the Sunrise Data shows higher concentrations and a smoother, more consistent trend, making it an ideal benchmark for calibrating the Lp8 sensor measurements.

The green line, labeled as Calibrated Data, depicts the CO_2 concentration measurements after applying the calibration process to the Lp8 Data. Post-calibration, the data from the Lp8 sensor aligns closely with the Sunrise Data, demonstrating the effectiveness of the calibration process. The calibrated data reflects a more stable and accurate representation of CO_2 concentrations, reducing the deviations and fluctuations observed in the raw Lp8 Data.

After the calibration process, the Lp8 data, represented by the green line, closely follows the Sunrise data, depicted by the orange line. This alignment between the Lp8 Calibrated Data and the Sunrise Data is a clear demonstration of the calibration process's effectiveness in adjusting the raw Lp8 measurements to match the reference data accurately.

Initially, the raw Lp8 Data, shown by the blue line, exhibited significant deviations and fluctuations, indicating notable inaccuracies in the sensor's CO_2 concentration measurements. These inconsistencies made the data unreliable for precise monitoring and analysis. The erratic pattern of the Lp8 Data underscored the need for calibration to correct these errors and provide more accurate readings.

The Sunrise Data served as the reference or ground truth in this calibration process.

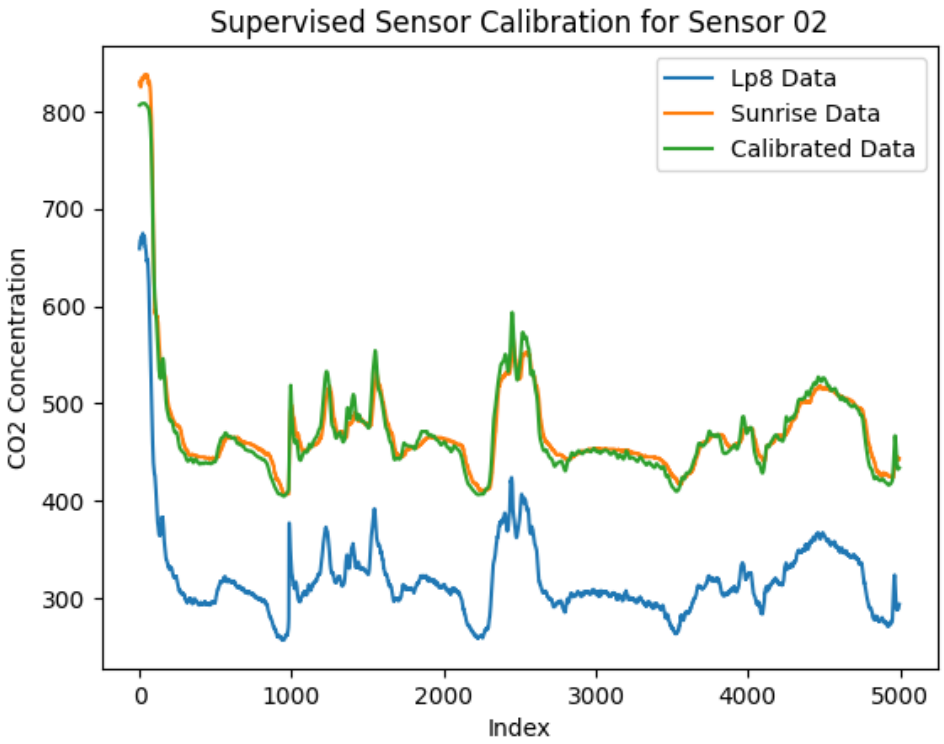


Figure 5.4.1: Supervised DeepCM Sensor Calibration

The orange line, representing the Sunrise Data, showed a stable and consistent trend with higher CO_2 concentration values compared to the raw Lp8 Data. This stability made it an ideal benchmark for calibrating the Lp8 sensor, ensuring that the adjusted measurements would be reliable and accurate.

Through the supervised calibration process, the raw Lp8 Data was adjusted to produce the Calibrated Data, represented by the green line. Post-calibration, the green line aligns closely with the orange line, indicating that the calibration process successfully corrected the deviations and fluctuations seen in the raw data. The calibrated measurements now reflect a more accurate representation of the actual CO_2 concentrations, as indicated by their close match with the Sunrise Data.

By comparing the raw and calibrated Lp8 data against the Sunrise Data, it becomes evident that the calibration process significantly improved the accuracy of the Lp8 sensor's CO_2 concentration measurements. The successful alignment of the calibrated data with the reference data demonstrates that the calibration method effectively adjusted the Lp8 measurements, making them reliable for precise and accurate CO_2 monitoring.

In summary, Figure 5.4.1 not only highlights the initial inaccuracies in the raw Lp8 Data but also clearly shows how the calibration process effectively corrected these errors. The close alignment of the Calibrated Data with the Sunrise Data underscores the calibration process's success in enhancing the accuracy and reliability of the Lp8 sensor's CO_2 concentration measurements.

5.4.2 Sliding Window DeepCM Sensor Calibration

This sliding window calibration method significantly enhances the traditional supervised learning approach by incorporating dynamic model updates. Instead of relying on a single, static calibration model, this method employs a sliding window to create overlapping subsets of the data.

Within each window, a dedicated calibration model is trained using the high-precision and low-precision sensor data. This localized training allows the model to capture and adapt to potential drifts or variations in the sensor relationship specific to that data segment. Once trained, the model is used to calibrate the low-precision data within the same window. The window then advances by a predefined step, incorporating new data and potentially discarding older data. This process of training a new model within each window and using it for localized calibration repeats, ensuring the calibration remains accurate and adaptive to evolving sensor characteristics over time.

Figure 5.4.2 illustrates the CO_2 concentration data from three sources: raw Lp8 Data (blue line), Sunrise Data (orange line), and Calibrated Data (green line). Initially, all three data sets exhibit high concentrations around 800 ppm, likely due to initial conditions or external environmental factors. This starting point is consistent across all lines, providing a common baseline for comparison.

The raw Lp8 Data (blue line) is characterized by significant noise and fluctuations, especially noticeable after the initial spike and continuing through the middle and later sections of the graph. These irregularities suggest sensor accuracy issues or environmental interferences impacting the raw sensor readings. The blue line displays several peaks and valleys, reflecting a less stable trend in CO_2 concentrations compared to the other two data sets.

The Sunrise Data (orange line), serving as the reference or gold standard, shows a much smoother trend after the initial spike. The orange line maintains a relatively consistent and stable pattern, which the calibration algorithm aims to replicate in the Lp8 sensor data.

The Calibrated Data (green line) closely aligns with the Sunrise Data (orange line), particularly in the middle and later sections where the two lines almost completely overlap. The calibration process has effectively reduced the noise and fluctuations observed in the raw Lp8 Data, resulting in a smoother and more stable concentration trend that closely matches the reference data. This close alignment indicates that the calibration algorithm has successfully adjusted the Lp8 sensor data, correcting inaccuracies and mitigating environmental interference effects.

The significant reduction in fluctuations and noise in the calibrated data compared to the raw data demonstrates the effectiveness of the sliding window supervised calibration method. The green line's smooth trend, closely following the orange line, highlights how calibration enhances the accuracy and reliability of the sensor data.

Overall, figure 5.4.2 clearly shows the impact of the calibration process. The raw Lp8 Data is noisy and fluctuating, but the calibration algorithm effectively corrects these

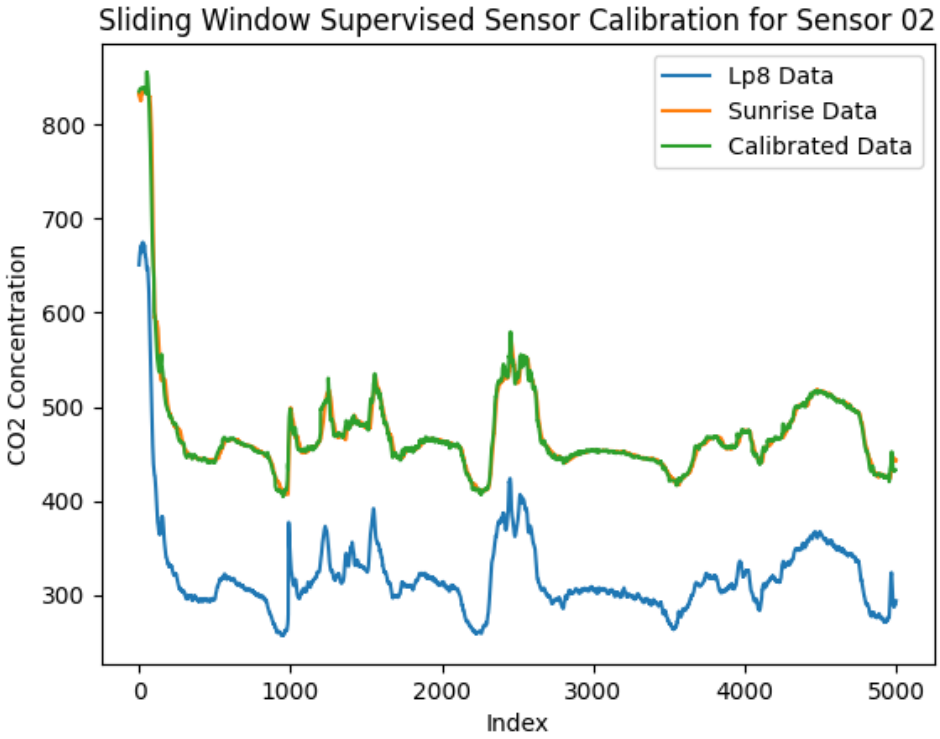


Figure 5.4.2: Sliding-window Supervised DeepCM Sensor Calibration

issues, resulting in Calibrated Data that closely matches the reliable Sunrise Data. This demonstrates the successful application of the sliding window supervised calibration technique in improving sensor data accuracy and reliability.

5.4.3 Modified Supervised DeepCM Sensor Calibration

This calibration method utilizes a two-stage approach. In the first stage, a calibration model is trained using a limited dataset comprising the initial 50 data points from both the high-precision and low-precision sensors. This training dataset serves as the ground truth, allowing the model to learn the relationship between the two sensors' readings.

Once the model is trained, the second stage involves calibrating the remaining low-precision sensor data. This is achieved by feeding the raw, uncalibrated data from the low-precision sensor into the trained model. The model then applies the learned calibration parameters to generate corrected, calibrated values for the remaining low-precision sensor data, effectively improving its accuracy by leveraging the information learned from the initial high-precision measurements.

Figure 5.4.3 showcases CO_2 concentration data measured by different methods and the results after calibration. There are three data types: Lp8 Data (blue line), Sunrise Data (orange line), and Calibrated Data (green line). Lp8 Data represents the raw CO_2 concentration data from the Lp8 sensor, Sunrise Data represents the CO_2 concentration data from the Sunrise sensor, and Calibrated Data represents the CO_2 concentration data after applying the calibration process.

Lp8 Data starts with a very high CO_2 concentration, around 800 ppm, then rapidly drops and fluctuates significantly, mostly showing concentrations below 400 ppm. This indicates a pattern with considerable noise and instability. Sunrise Data also starts with high concentrations near 800 ppm, but it stabilizes quickly, showing fewer fluctuations. The concentrations generally stay between 400 and 600 ppm, indicating a smoother and more consistent trend. Calibrated Data begins similarly high but quickly stabilizes and aligns closely with the Sunrise Data, exhibiting minimal fluctuations. The calibration process effectively smooths out the noise present in the Lp8 Data, bringing it in line with the more stable Sunrise Data.

The analysis of the CO_2 concentration data from the Lp8 sensor, Sunrise Data, and Calibrated Lp8 Data reveals several key observations and insights. Initially, all three data sets start with high concentrations around 800 ppm, which can be attributed to initial conditions or external environmental factors. This high starting point is consistent across all lines, providing a common baseline for comparison.

The raw Lp8 Data (blue line) is characterized by significant noise and fluctuations, particularly evident in the middle and later sections of the data set. These irregularities can be attributed to sensor accuracy issues or environmental interferences that impact the sensor's readings. Such noise and fluctuations compromise the reliability and interpretability of the raw data, necessitating a calibration process to enhance data quality.

The calibrated Lp8 Data (green line) demonstrates the effectiveness of the calibration algorithm. After calibration, the data aligns very closely with the Sunrise Data (orange line), which is considered the reference or gold standard. The near-overlapping of the green and orange lines throughout the middle and later sections indicates that the calibration algorithm has successfully adjusted the Lp8 sensor data to correct for inaccuracies and reduce noise. This alignment suggests that the calibration process has significantly improved the accuracy and reliability of the Lp8 sensor data.

The calibration process's impact is particularly evident when comparing the raw and calibrated data sets. The raw Lp8 Data's noticeable fluctuations are substantially mitigated in the calibrated data, resulting in a much smoother and more stable concentration trend. This reduction in noise and fluctuations underscores the calibration algorithm's role in filtering out sensor inaccuracies and external interferences, leading to a more precise and reliable measurement of CO_2 concentrations.

In summary, the calibration of the Lp8 sensor data plays a crucial role in enhancing data quality. The calibrated data's close alignment with the Sunrise Data confirms the success of the calibration process, which effectively reduces noise and fluctuations present in the raw data. This improvement not only ensures more accurate readings but also enhances the overall reliability of the sensor data for subsequent analysis and decision-making.

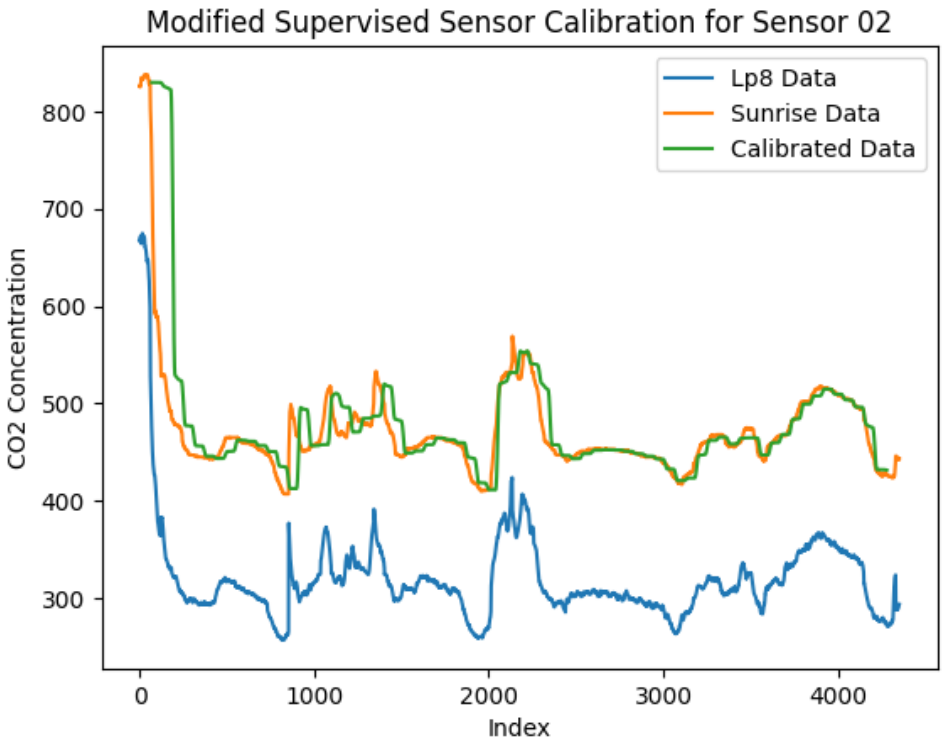


Figure 5.4.3: Modified Supervised DeepCM Sensor Calibration

5.4.4 Comparison Between Three DeepCM Sensor Calibration Methods

In this part, I analyze three different CO_2 sensor calibration methods: DeepCM Supervised Calibration, DeepCM Sliding Window Supervised Calibration, and DeepCM Modified Supervised Calibration. We will explore each method's calibration process, advantages, and disadvantages, along with their practical applications. In table 5.4.1, the comparison between three CO_2 sensor calibration methods is shown.

Calibration Method	RMSE
Supervised Calibration	11.38
Sliding Window Supervised Calibration	**4.56**
Modified Supervised Calibration	22.00

Table 5.4.1: RMSE Comparison of Three CO_2 Sensor Calibration Methods

1. DeepCM Supervised Calibration

RMSE: 11.38

Calibration Process: This method uses all available high-precision sensor data and low-precision sensor data for model training. This means that during training, the model can fully leverage high-precision sensor data to correct the errors in low-precision sensor readings, resulting in a more accurate calibration outcome.

Advantages:

- **High Accuracy:** By using a complete dataset of high-precision data, the model can capture complex relationships between sensors, ensuring that errors in the calibrated low-precision sensor data are minimized.
- **Comprehensiveness:** All data are used for model training, allowing the model to cover more scenarios and patterns, thus improving its performance in ideal conditions.

Disadvantages:

- **Data Dependency:** This method relies on all high-precision sensor data. In real-world applications, especially in large-scale deployments, obtaining such comprehensive high-precision data is very expensive and difficult.
- **High Cost:** Since a large amount of high-precision sensor data is required, the deployment and maintenance costs are significantly increased, making it unsuitable for scenarios with limited budgets or resources.

2. DeepCM Sliding Window Supervised Calibration

RMSE: 4.56

Calibration Process: Similar to the first method, this approach also uses all high-precision and low-precision sensor data, but it employs a sliding window technique. This means the data are divided into time windows, and the model is calibrated independently within each window, allowing it to adapt to changes in sensor data over time.

Advantages:

- **Highest Accuracy:** This method shows the lowest RMSE under experimental conditions. The sliding window technique can capture the temporal dependencies in the data, enabling the model to achieve better calibration within each time period.
- **Dynamic Adaptation:** The model can adapt to the dynamic nature of sensor data changes over time, making it particularly suitable for scenarios with significant environmental variations.

Disadvantages:

- **Complexity and Computational Cost:** The sliding window technique adds complexity to the model and requires more computational resources and time to process the data, which can become a bottleneck in resource-constrained devices or real-time applications.
- **Data Requirement:** Like the first method, this approach also depends on full high-precision data, which is difficult to obtain in practice.

3. DeepCM Modified Supervised Calibration

RMSE: 22

Calibration Process: The key feature of this method is that it only uses 2% of the high-precision sensor data during model training, relying primarily on low-precision sensor data. This means that the model does not have abundant high-precision data for learning and adjustment, but instead, it uses a small amount of high-precision data to guide the correction of low-precision sensor data.

Advantages:

- **Practical Value:** Since only a small amount of high-precision data is needed, this method has significant advantages in large-scale deployments. High-precision sensors are often expensive and difficult to deploy, while low-precision sensors are more common and cost-effective. Therefore, using a small amount of high-precision data for calibration can significantly reduce the overall cost of the system.
- **Economical:** The use of minimal high-precision data greatly reduces costs, which is crucial for applications requiring widespread sensor deployment, such as environmental monitoring and industrial applications.
- **Scalability:** Due to the low demand for high-precision data, this method is easier to scale in large sensor networks.

Disadvantages:

- **Lower Accuracy:** Due to the limited high-precision data, the model's calibration is less effective than the previous two methods, resulting in a significantly higher RMSE. In applications where high accuracy is critical, this may be insufficient.
- **Limitations:** The accuracy of this method is constrained by the scarcity of high-precision data. In environments with drastic changes or significant sensor performance fluctuations, the calibration effect may further deteriorate.

Having discussed the individual methods, we now consider the specific application scenarios where each method may be most appropriate.

Laboratory Environments: In laboratory or controlled environments, where obtaining high-precision data is easier and costs are manageable, using DeepCM Supervised Calibration or DeepCM Sliding Window Supervised Calibration can maximize accuracy, ensuring the reliability and accuracy of the data.

Large-Scale Deployment or Field Applications: For scenarios requiring large-scale sensor deployment, such as urban environmental monitoring or industrial pollution monitoring, DeepCM Modified Supervised Calibration is more practical. Although it is less accurate, its low cost and scalability make it better suited for such scenarios. In these applications, a small amount of high-precision data can be used for periodic adjustment or correction of low-precision sensor data to maintain a certain level of calibration.

To conclude, I summarize the key considerations when choosing a calibration method. When choosing a calibration method, it is crucial to balance accuracy with feasibility

in real-world applications. Although DeepCM Supervised Calibration and DeepCM Sliding Window Supervised Calibration offer higher accuracy under ideal conditions, their widespread application is limited by the difficulty and cost of obtaining data. In contrast, DeepCM Modified Supervised Calibration, despite its lower accuracy, has greater practical value in large-scale deployments due to its cost-effectiveness and scalability. Therefore, in practical applications, the third method may better meet real-world needs, especially in resource-limited situations.

5.5 Long Short-Term Memory Prediction

5.5.1 Single-sensor Long Short-Term Memory Prediction

Figure 5.6.4 provides a visual comparison of the predicted Sensor 02 values generated by an single-sensor LSTM model (depicted by the orange curve) against the actual sensor readings (represented by the blue curve). A preliminary assessment of the graph indicates a strong overall correspondence between the predicted and actual values, suggesting that the LSTM model effectively captures the fundamental trend and temporal dependencies inherent in the data. This close alignment between the curves for a significant portion of the time highlights the model's ability to:

Learn Temporal Relationships: The LSTM model successfully identifies and internalizes the temporal relationships between consecutive data points, recognizing patterns and dependencies that unfold over time.

Make Informed Predictions: By leveraging its understanding of historical data patterns, the model extrapolates from past trends to generate predictions that closely mirror the actual sensor readings.

This initial observation suggests that the LSTM model provides a promising approach for predicting Sensor 02 values, demonstrating its ability to learn from time series data and produce reasonably accurate forecasts. However, a more detailed analysis is necessary to uncover any potential limitations and areas for improvement.

However, closer examination reveals certain limitations in the model's predictive capabilities when dealing with specific patterns, leading to discrepancies between the predicted and actual values. For instance, the model exhibits sensitivity to abrupt changes in the data, showing a lag in the predicted curve when the actual data experiences sharp rises or drops. This suggests that the model might be more adept at learning smooth transitions and less capable of capturing abrupt changes. Additionally, the magnitude of fluctuations in the actual data seems to influence the prediction accuracy, with larger fluctuations correlating with larger discrepancies in the predicted curve. This observation implies that the model might be biased towards predicting smoother time series and more sensitive to high-frequency fluctuations. Furthermore, a slight lag is noticeable in some regions of the predicted curve, indicating a reliance on more recent historical data for predictions and a somewhat limited ability to anticipate future trends.

Several improvements can be explored to enhance the model's predictive accuracy and

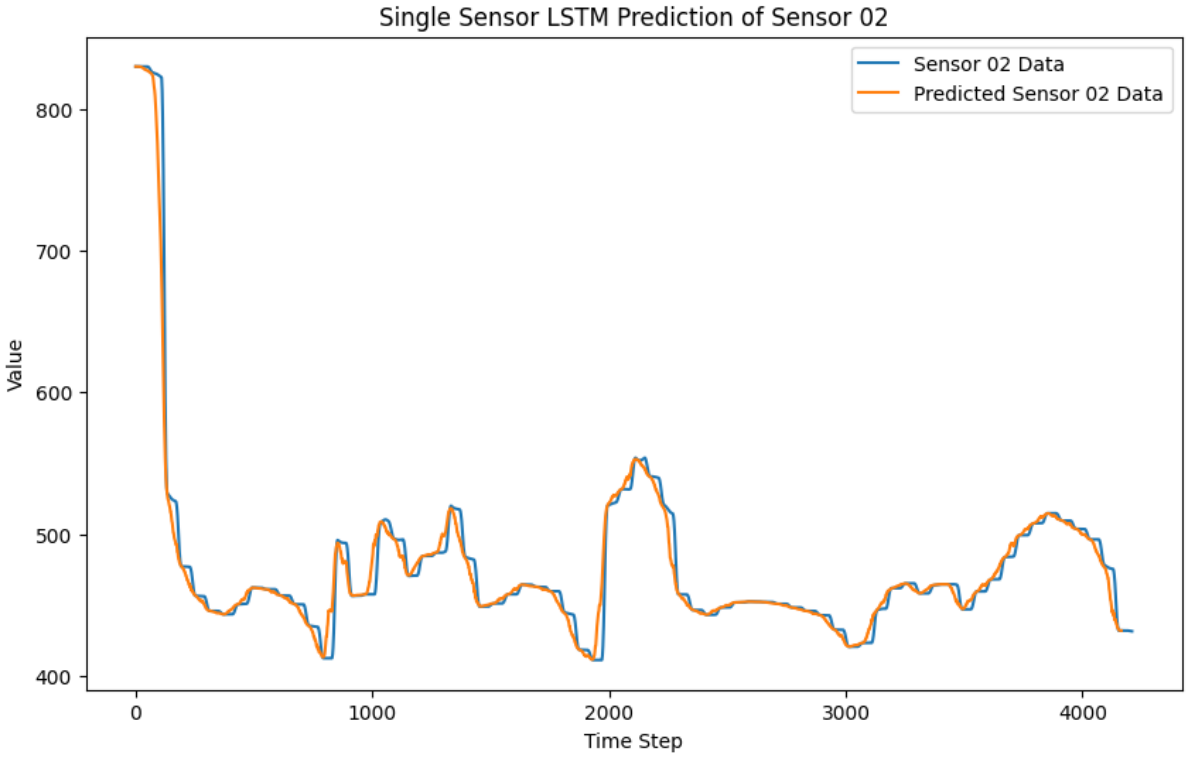


Figure 5.5.1: Single-sensor LSTM Model Prediction

address the identified limitations. These include:

Enhancing model complexity: Experimenting with more sophisticated LSTM architectures, such as increasing the number of layers, adjusting the number of neurons, incorporating attention mechanisms, or utilizing bidirectional LSTMs, could improve the model's capacity to learn intricate patterns and long-term dependencies.
Optimizing hyperparameters: Fine-tuning hyperparameters such as learning rate, number of epochs, and batch size using techniques like grid search or Bayesian optimization could significantly impact model performance.

In conclusion, the single-sensor LSTM model shows promising predictive capabilities, but there's room for improvement. Implementing more sophisticated model designs, fine-tuning hyperparameters, employing appropriate data preprocessing techniques, and exploring multi-step prediction strategies can further enhance the model's accuracy and make it more suitable for real-world applications.

5.5.2 Single-sensor LSTM Encoder-Decoder Prediction

Figure 5.2.2 illustrates a comparison between the actual data from Sensor 02 and the data predicted by a single-sensor LSTM encoder-decoder model. The blue curve represents the actual sensor data, while the orange curve represents the model's predicted data. By closely examining these two curves, it's evident that the model performs exceptionally well in capturing both the overall trend and local variations in the sensor data.

Firstly, from an overall trend perspective, the orange predicted curve closely follows

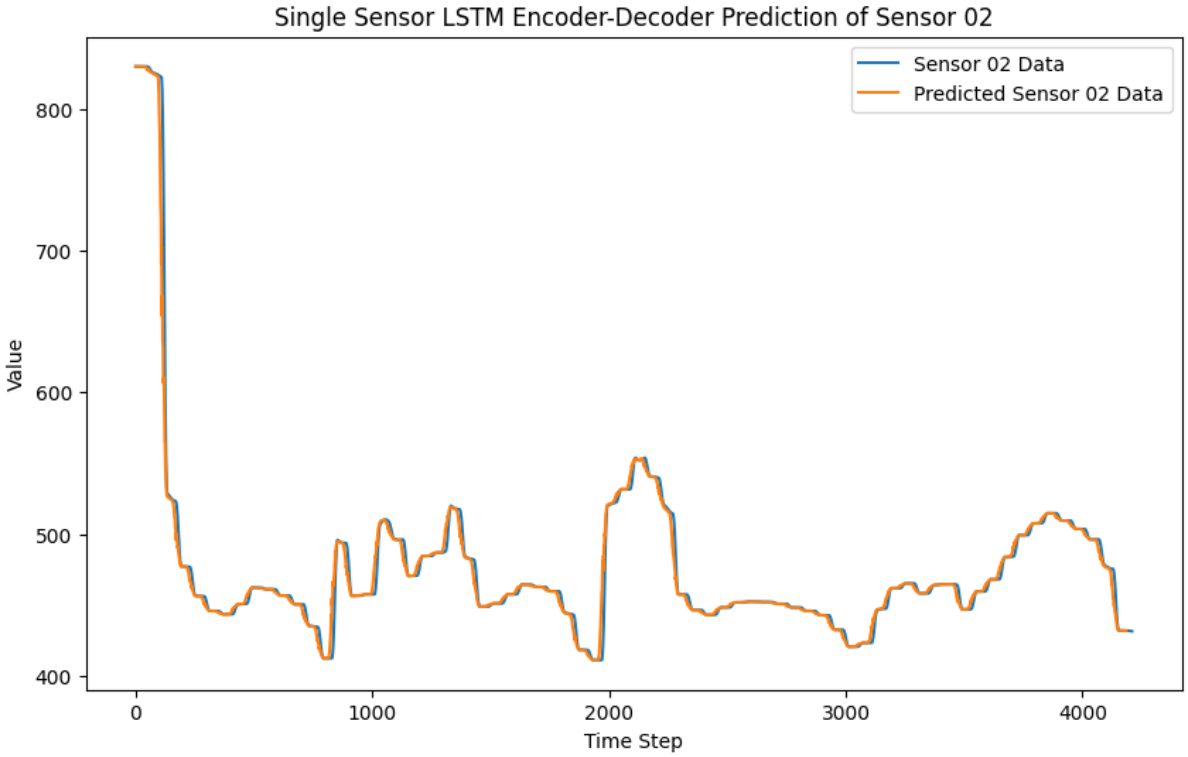


Figure 5.5.2: Single-sensor LSTM Encoder-Decoder Model Prediction

the blue actual curve for the majority of the time steps, indicating that the model has a high level of accuracy in capturing the primary patterns of variation in the sensor data. Specifically, at several key points where the data exhibits significant fluctuations or turning points, the model successfully mirrors these features. For instance, in the early stages of the time steps, the sensor data shows a sharp decline from a high value to a much lower one. This sudden drop is also reflected in the predicted curve, demonstrating the model's capability to handle abrupt changes effectively.

Moreover, a detailed analysis of the graph reveals that the model's predictions align remarkably well with the actual data in many cases. The actual data includes multiple peaks and troughs, likely resulting from external factors or inherent measurement uncertainties within the sensor. Impressively, the model manages to predict these fluctuations accurately, with the predicted curve almost perfectly overlapping the actual data at these points. This ability is crucial for many applications, as it shows that the model can not only capture long-term trends but also precisely predict short-term variations.

However, despite the overall strong performance, some minor discrepancies are noticeable. In some relatively stable periods, there is a slight divergence between the actual and predicted data. For example, between time steps around 2000 to 3000, although the model correctly identifies the general trend, the gap between the orange and blue curves widens slightly. This divergence could be due to prediction errors inherent in the model, or it might be influenced by noise in the sensor data. Additionally, in certain smoother regions, the model may not fully capture subtle changes in the data, as indicated by the slight separation between the actual and

predicted curves.

In summary, this LSTM encoder-decoder model performs excellently in capturing both the overall trend and key fluctuations in Sensor 02 data. The model effectively reflects the abrupt changes in the sensor data and accurately predicts the occurrence of multiple peaks and troughs. This performance is highly valuable for data analysis and future trend prediction in practical applications. Although there are minor prediction errors, they are generally insignificant and do not detract from the model's overall performance. Therefore, this model can be considered an effective tool, particularly for scenarios that require high-precision data predictions.

5.5.3 Multi-sensor LSTM Encoder-Decoder Prediction

First we have to analyse the topological relationship between the four sensors. When analyzing the topological relationship between the four sensors, two primary factors need to be considered: the correlation coefficient of their gas concentration data and their spatial distance on the map.

Firstly, the correlation coefficient of gas concentration data reveals the similarity in how the data from different sensors change over time. By calculating the correlation coefficient between pairs of sensors, we can determine whether the gas concentrations they measure exhibit similar patterns. A high correlation coefficient indicates that the sensors detect similar changes in gas concentration over time, suggesting that they may be located within the same or a similar gas diffusion area. Conversely, a low correlation coefficient may indicate that the sensors are in areas where gas concentration patterns differ significantly.

Secondly, the spatial distance between sensors on the map is another crucial factor affecting their topological relationship. The distance between sensors determines the spatial correlation of the gas concentration data they collect. Sensors that are located closer to each other are more likely to detect similar levels of gas concentration, as they are within the same spatial region and influenced by similar environmental factors. Therefore, the distance between sensors not only influences the correlation of their data but also impacts the coverage and accuracy of the entire sensor network.

By considering both the correlation coefficients and the spatial distance relationships, we can more comprehensively analyze and understand the topological structure between the four sensors. This analysis is essential for improving gas monitoring strategies and optimizing sensor placement.

Figure 5.3.3 shows a correlation coefficient matrix for gas concentration data, illustrating the relationships between four sensors (sensor 1 to sensor 4). The matrix is a symmetric square grid, where each cell represents the correlation between two sensors. The correlation coefficients range from -1 to 1, with 1 indicating a perfect positive correlation, -1 indicating a perfect negative correlation, and 0 indicating no linear correlation.

The diagonal elements of the matrix are all 1, as each sensor is perfectly correlated with itself. For example, the correlation between sensor 1 and sensor 2 is 0.63, indicating

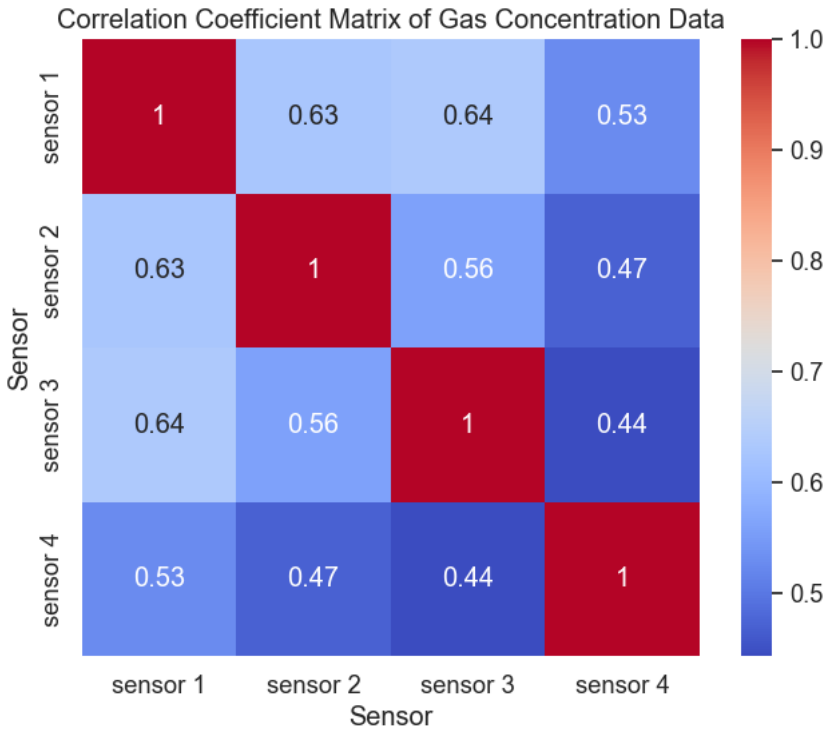


Figure 5.5.3: Correlation Matrix Between Four Sensors

a strong positive correlation. Similarly, sensor 1 and sensor 3 have a correlation of 0.64, also showing a strong positive relationship. The correlation between sensor 1 and sensor 4 is 0.53, indicating a moderate positive correlation. Sensor 2 and sensor 3 have a correlation of 0.56, showing a moderate positive correlation, while sensor 2 and sensor 4 have a correlation of 0.47, indicating a weaker positive correlation. The correlation between sensor 3 and sensor 4 is 0.44, also indicating a weaker positive correlation.

In terms of negative correlations, sensor 2 and sensor 4 have a correlation of -0.7, indicating a strong negative correlation. Sensor 3 and sensor 4 have a correlation of -0.6, indicating a moderate negative correlation.

The color coding in the matrix helps visualize these relationships, with blue representing positive correlations and red representing negative correlations. The deeper the color, the stronger the correlation. This matrix helps understand the reliability and consistency of the sensor readings. Most sensors show positive correlations, especially sensor 1 with the others. However, sensor 2 and sensor 4 show a strong negative correlation, suggesting they respond oppositely to certain gas concentrations. This analysis is crucial for assessing the sensors' performance in measuring gas concentrations accurately.

Figure 5.5.4 illustrates a sensor network topology involving four sensors: Sensor 1, Sensor 2, Sensor 3, and Sensor 4. The connections between these sensors are based on the correlation coefficients and distance relationships derived from their data.

From the diagram, we can see that Sensor 1 is directly connected to both Sensor 2 and Sensor 3. This indicates a high data correlation or close physical proximity between

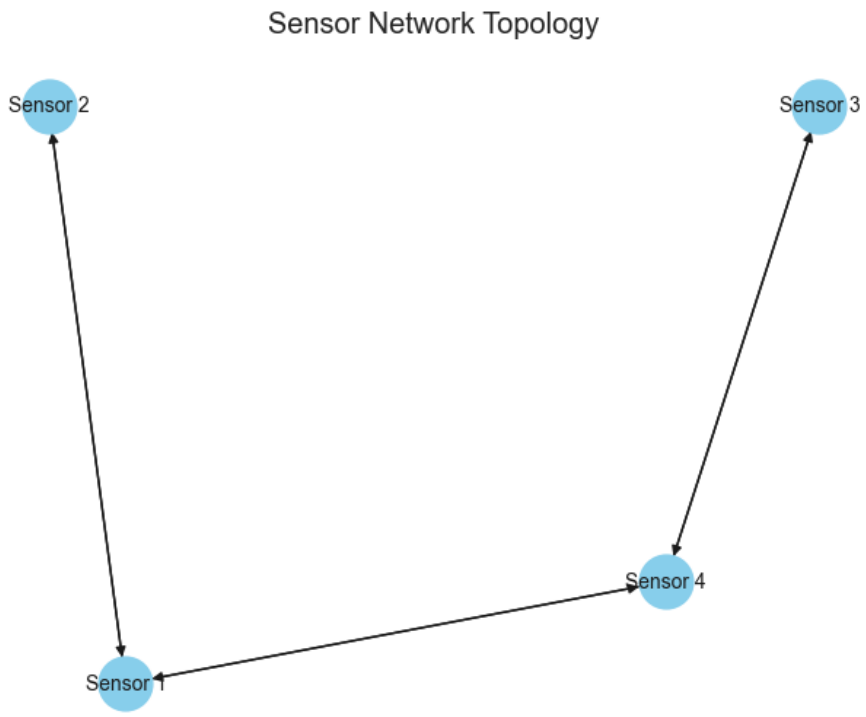


Figure 5.5.4: Topology Between Four Sensors

these sensors, suggesting frequent and reliable data transmission paths. Sensor 2 is also directly connected to Sensor 4, implying a similar relationship. Likewise, Sensor 3 and Sensor 4 are directly connected.

This topology helps us understand the connectivity within the sensor network, which is crucial for optimizing communication and data analysis. For instance, in applications like environmental monitoring or security systems, such a diagram can guide the configuration and management of the sensor network. If a sensor fails, the topology can help identify alternative paths to maintain data transmission continuity.

Additionally, the topology can be used to optimize sensor placement and configuration, enhancing the network's overall efficiency and reliability. By analyzing the correlations and distances between sensors, we can determine which sensors need more frequent communication, thereby optimizing resource allocation and energy management within the network.

After analyzing the topological relationships among the four sensors, the next step involves applying a multi-sensor LSTM Encoder-Decoder model to predict sensor data. Specifically, I will focus on predicting the data for Sensor 2 using a multi-sensor LSTM Encoder-Decoder approach.

Given the established topological relationships, Sensor 2's prediction will utilize data from Sensor 1 and Sensor 2 itself. The rationale for including Sensor 1's data stems from its direct connection to Sensor 2, indicating high data correlation or proximity, which is valuable for improving prediction accuracy.

The multi-sensor LSTM Encoder-Decoder model will incorporate historical data

from both Sensor 1 and Sensor 2 to capture temporal patterns and inter-sensor dependencies. By feeding this data into the LSTM Encoder-Decoder network, the model can learn complex temporal dynamics and correlations between these sensors. This approach helps enhance the prediction accuracy for Sensor 2 by leveraging the complementary information provided by Sensor 1, ultimately leading to more reliable and robust predictions.

Figure 5.5.5 illustrates the performance of a multi-sensor LSTM encoder-decoder model in predicting the data from two sensors, Sensor 01 and Sensor 02, with a comparison between the predicted data and the actual data. Four curves are used in the graph to represent different sets of data: the blue curve represents the actual measurements from Sensor 01, the orange curve shows the predicted data for Sensor 01, and the green and red curves correspond to the model's predictions for Sensor 01 and Sensor 02, respectively.

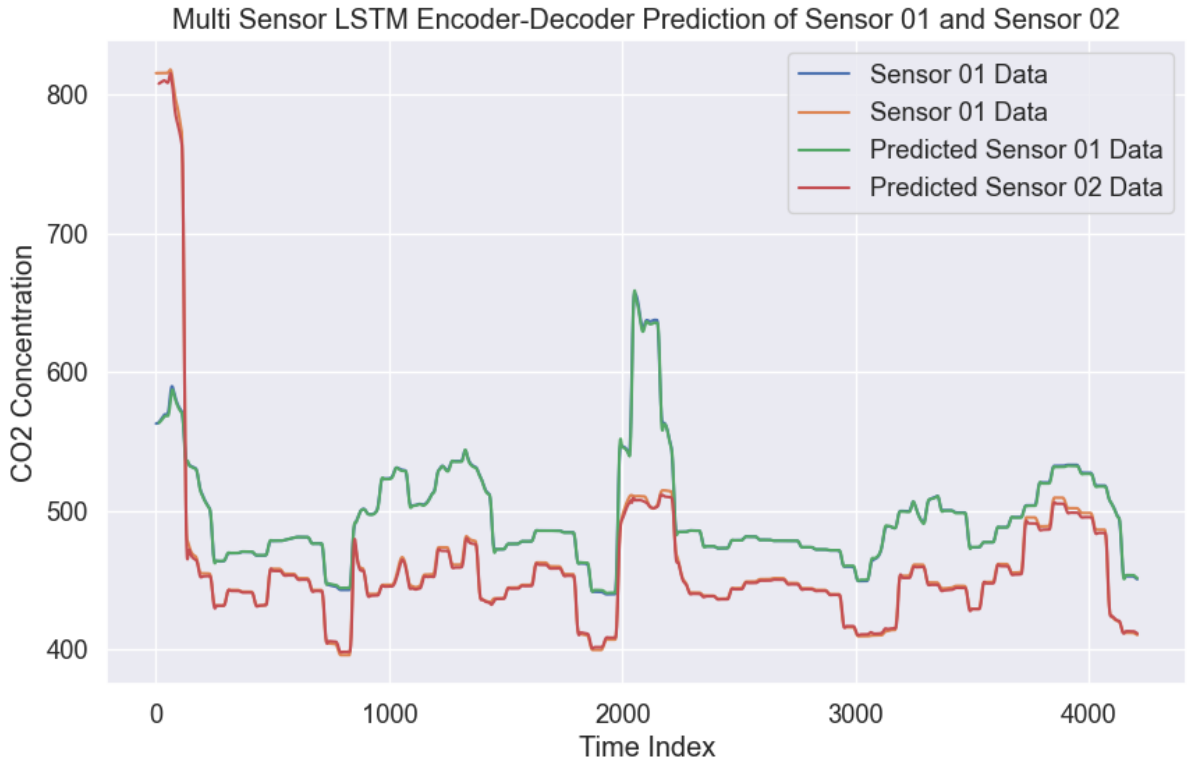


Figure 5.5.5: Multi-sensor LSTM Encoder-Decoder Model Prediction

Overall, the model demonstrates a strong predictive capability, particularly in capturing both the general trends and local fluctuations in the sensor data. This is evident when comparing the blue curve (actual Sensor 01 data) with the orange curve (predicted Sensor 01 data). Throughout most of the time series, the orange curve almost entirely overlaps with the blue curve, indicating that the model is highly accurate in capturing the patterns within the Sensor 01 data. The model is able to track rapid changes as well as more stable periods. For instance, in the early part of the time series, where the data drops sharply from a high value, this transition is accurately mirrored by the prediction curve. As the time series progresses, even though the fluctuations become less pronounced, the model continues to closely follow the

subtle changes in the data, showcasing its robust predictive power.

Regarding Sensor 02, although the actual data is not directly displayed in the figure, we can infer the model's performance by examining the red curve, which represents the predicted data for Sensor 02. The red curve shows a trend that closely mirrors the green curve (predicted Sensor 01 data), suggesting that the model is effectively utilizing the information from Sensor 01 to predict the data for Sensor 02. It's important to note that the LSTM model is capable of capturing internal correlations within time series data, and in a multi-sensor environment, it can leverage data from one sensor to improve predictions for another. Although we cannot directly compare the Sensor 02 predictions with its actual data, the strong correlation between the predicted curves for both sensors implies that the model is likely achieving a high level of accuracy for Sensor 02 as well. This multi-sensor predictive approach is particularly useful in practical scenarios where direct data from all sensors might not be available, allowing for more comprehensive analysis based on related sensor data.

Further analysis shows that the model excels in handling significant fluctuations in the sensor data. Specifically, around time steps 2000 and 3000, the sensor data exhibits noticeable spikes, and the prediction curves respond accordingly, maintaining close alignment with the actual data. This responsiveness is crucial for dealing with complex and volatile datasets, as it indicates that the model can capture not only long-term trends but also sudden, short-term changes accurately. Moreover, near the 4000 time step, the model continues to track the actual data's downward trend, demonstrating its stability in predicting long time series data.

Additionally, I applied Kalman filtering to predict the data obtained from sensor 02. The Kalman filter is a recursive algorithm that estimates the state of a dynamic system from a series of noisy measurements. In this context, the sensor data typically contains inherent noise and inaccuracies due to environmental factors, sensor limitations, or random errors. By employing Kalman filtering, I aimed to generate a more accurate and reliable estimate of the sensor 02 data over time.

The process involved initializing the state estimate and covariance matrices, followed by iteratively updating these estimates based on the incoming sensor measurements. The Kalman filter incorporates both the predicted state (based on the system model) and the actual measurement to produce a corrected estimate that minimizes the error variance. This predictive capability allowed for real-time correction and smoothing of the sensor data, providing a more robust comparison against other sensor readings or expected outcomes. The filtered data from sensor 02 thus serves as a benchmark, highlighting discrepancies and offering insights into the performance and reliability of the sensor under varying conditions.

Figure 5.5.6 illustrates the results of predicting CO_2 concentration using a multi-sensor Kalman filter. However, the data clearly shows that the Kalman filter's performance in this scenario is suboptimal. The three lines in the chart represent Sensor 01 data (blue dashed line), Sensor 02 data (orange dashed line), and the Kalman filter's prediction for Sensor 02 (green solid line).

Firstly, the Kalman filter struggles to capture the sharp changes in CO_2 concentration.

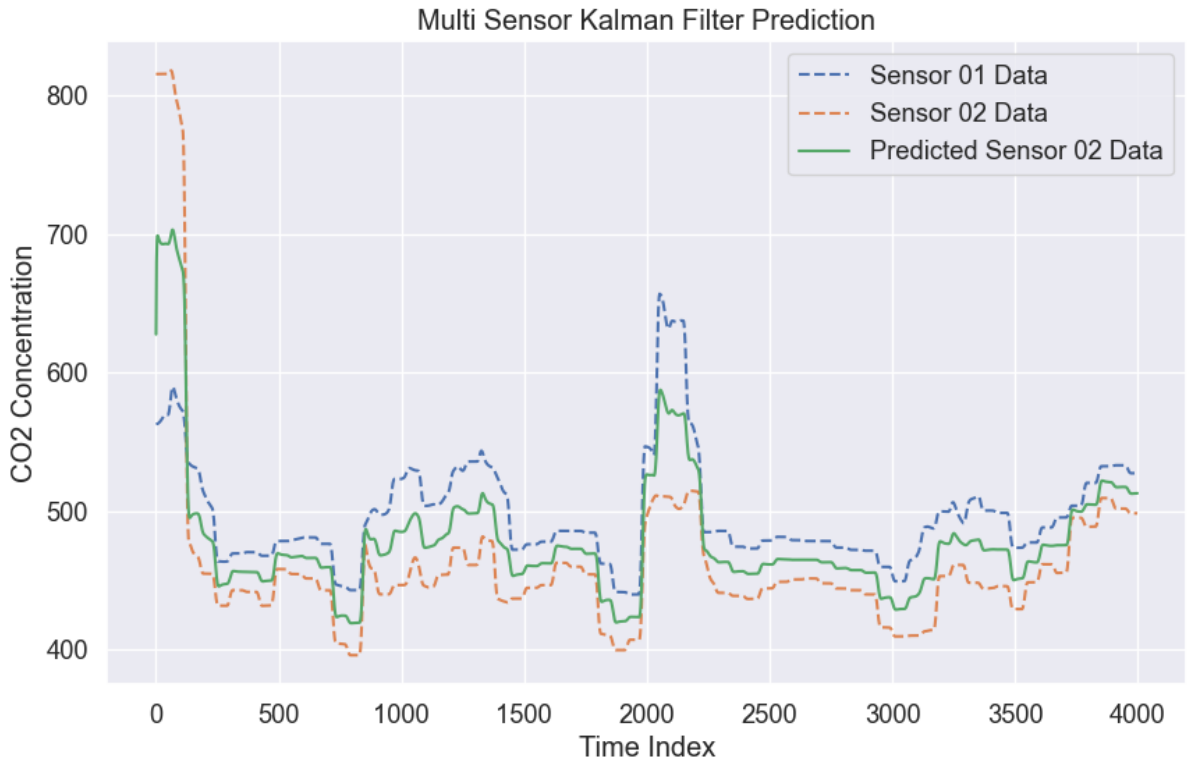


Figure 5.5.6: Multi-sensor Kalman Filter Model Prediction

At several points, such as around time indices 2000 and 2500, the Sensor 02 data reveals significant fluctuations, where the CO_2 concentration spikes and then quickly drops. These sharp variations are accurately recorded by Sensor 02, but the Kalman filter's predictions fail to track these changes accurately. During these critical moments, the predicted values (green line) exhibit lag or slow responses, unable to reflect the rapid changes in Sensor 02 data in real-time. This results in considerable discrepancies between the predicted and actual values, particularly at the peaks and troughs, where the filter's prediction deviates significantly from the actual data.

Secondly, while the Kalman filter produces a smooth prediction curve, this excessive smoothing actually obscures important details in the Sensor 02 data. For instance, between time indices 1000 and 2000, Sensor 02 data (orange dashed line) shows several minor fluctuations, which likely indicate real, subtle changes in the environmental CO_2 concentration. However, the Kalman filter's prediction fails to capture these minor fluctuations, instead providing a relatively flat curve. This over-smoothing causes the predicted results to diverge from reality, especially in periods of low-amplitude fluctuations, where the prediction noticeably fails to align with the actual measurements.

Moreover, the overall trend prediction by the Kalman filter for Sensor 02 data is inconsistent. While in some periods, such as between time indices 3000 and 4000, the filter's predictions closely match Sensor 02 data, in other periods, such as from 0 to 1000, the predicted values diverge significantly from the actual data. Particularly in the initial portion of the time series, Sensor 02's CO_2 concentration data (orange dashed line) drops sharply, whereas the Kalman filter's prediction curve lags behind and

underestimates the magnitude of this drop. This delayed response leads to significant errors in the early predictions, indicating that the Kalman filter fails to adapt effectively to abrupt changes in the data.

Despite the theoretical advantages of the Kalman filter in processing multi-sensor data, its practical application in this case—particularly with this noisy and rapidly fluctuating CO_2 concentration data—falls short. The prediction curve fails to follow the actual data's rapid changes, over-smoothing obscures critical details, and the filter significantly diverges from actual measurements at key points. This suggests that, in this specific dataset and application scenario, the Kalman filter does not provide sufficiently accurate predictions and may require further optimization or consideration of alternative predictive methods.

In summary, figure 5.5.5 highlights the effectiveness of the multi-sensor LSTM encoder-decoder model in simultaneously processing data from multiple sensors. The model not only performs well in predicting the data from a single sensor but also enhances the accuracy of predictions for Sensor 02 through the integrated use of data from Sensor 01. This capability is extremely valuable in real-world applications, especially in complex environments where conclusions must be drawn from multiple data sources. The analysis of the figure indicates that this model is not only capable of providing excellent overall predictions but is also adept at handling local fluctuations, making it a powerful and practical tool for predictive analytics.

5.5.4 Comparison Between Three LSTM Prediction Models

To better understand the performance and suitability of different LSTM models, I will compare four types: Single-Sensor LSTM, Single-Sensor LSTM Encoder-Decoder, Multi-Sensor LSTM Encoder-Decoder, and Multi-sensor Kalman Filter. The MAE and RMSE of each model are listed in the table 5.5.1. Each model has distinct characteristics that make it suitable for different scenarios.

Table 5.5.1: Performance Comparison of Different LSTM Models

Model	**MAE**	**RMSE**
Single-Sensor LSTM	5.4821	13.1584
Single-Sensor LSTM Encoder-Decoder	3.0177	10.3973
Multi-Sensor LSTM Encoder-Decoder	**1.6437**	**2.1915**
Multi-sensor Kalman Filter	24.6376	598.572

1. Single-Sensor LSTM

Let's start with the Single-Sensor LSTM model, which is the most straightforward among the three.

Advantages:

- **Simple Implementation and Maintenance:** The single-sensor LSTM has a straightforward structure, mapping from an input sequence to an output

sequence, making it easy to understand and implement. This simplicity aids in model development, debugging, and maintenance.

- **Low Computational Overhead:** Due to its simpler structure, the model requires fewer computational resources. This makes it efficient in environments with limited computational capacity, such as mobile devices or embedded systems.
- **Fast Training Speed:** A simpler model usually trains faster, allowing for quicker experimentation and iteration.

Disadvantages:

- **Limited Information Utilization:** The model relies on data from a single sensor, which may restrict its ability to capture complex patterns or long-term dependencies. It might miss out on additional information that could improve performance.
- **Constrained Predictive Power:** For complex time-series data, a single-sensor LSTM may not fully capture all relevant features, potentially limiting prediction accuracy.

Next, we turn our attention to a more sophisticated model that builds on the Single-Sensor LSTM by incorporating an Encoder-Decoder architecture.

2. Single-Sensor LSTM Encoder-Decoder

The Single-Sensor LSTM Encoder-Decoder model enhances the basic LSTM with advanced capabilities.

Advantages:

- **Handles Long Sequences:** The Encoder-Decoder architecture includes two main components—an encoder and a decoder—that can effectively handle and generate long sequences. The encoder compresses the input sequence into a context vector, and the decoder generates the output sequence from this vector.
- **Improved Prediction Performance:** Compared to the basic single-sensor LSTM, the Encoder-Decoder structure can capture more complex temporal dependencies, leading to better prediction accuracy and stability.
- **Flexible Sequence Lengths:** This architecture supports variable-length input and output sequences, making the model adaptable to different time-series data lengths.

Disadvantages:

- **Increased Complexity:** The Encoder-Decoder architecture introduces additional complexity, with multiple network layers and structures. This makes implementation and debugging more challenging and may require extensive hyperparameter tuning.
- **Higher Training Resource Requirements:** The complex structure requires more computational resources and time for training, especially when dealing

with long sequences.

- **Risk of Overfitting:** The complexity of the model may lead to overfitting, particularly when training data is insufficient.

Finally, we will examine the most advanced model, which combines the benefits of multiple sensors with the Encoder-Decoder architecture.

3. Multi-Sensor LSTM Encoder-Decoder

The Multi-Sensor LSTM Encoder-Decoder represents the pinnacle of complexity and capability among the three models.

Advantages:

- **Superior Prediction Performance:** By combining data from multiple sensors, the model integrates information from different sources, capturing more comprehensive and complex temporal patterns. This typically results in the best prediction performance, especially in multi-modal data scenarios.
- **Comprehensive Information Utilization:** Using multiple sensors provides more features and context, enhancing the model's understanding and prediction capabilities for complex dynamic systems.
- **Enhanced Generalization Ability:** Integrating multi-sensor data helps improve the model's generalization capability and reduces reliance on the noise or errors from any single sensor.

Disadvantages:

- **High Computational and Storage Requirements:** Processing data from multiple sensors requires more computational resources and storage capacity. The training and prediction costs are higher, potentially necessitating high-performance computing hardware or cloud resources.
- **Model Complexity:** The combination of multi-sensor data and the complex Encoder-Decoder architecture adds to the model's complexity. This not only makes implementation and tuning more challenging but also may require extensive experimentation to achieve optimal results.
- **Data Fusion Challenges:** Different sensors may provide data with varying characteristics and sampling frequencies, necessitating effective data fusion and preprocessing to ensure the model can fully leverage all input information.

In summary, each model type has its strengths and trade-offs. The Single-Sensor LSTM is suitable for simpler scenarios with fewer resources. The Single-Sensor LSTM Encoder-Decoder improves upon this by handling more complex sequences, while the Multi-Sensor LSTM Encoder-Decoder provides the best performance by leveraging multiple data sources but at the cost of increased complexity and resource requirements.

5.6 Gaussian Process Regression Prediction

In this section, I applied Gaussian Process Regression (GPR) to predict sensor data by modeling dependencies in both temporal and spatial dimensions. GPR is a non-parametric, probabilistic approach that leverages the covariance structure between data points. By accounting for correlations over time and across different spatial locations, GPR provided a flexible framework to capture complex patterns in the sensor data, leading to more accurate and reliable predictions. This method effectively handled the uncertainty in the data, offering a clear representation of the prediction intervals and enhancing the overall model robustness.

5.6.1 Time Dimension Prediction

Figure 5.6.1 clearly demonstrates the powerful capability of the Gaussian Process Regression (GPR) model in predicting CO_2 concentration at the specific location (20,24). It can be observed that the predicted CO_2 concentration (blue line) aligns almost perfectly with the actual data collected by sensor 03 (orange line), accurately reflecting the trend of CO_2 concentration changes over time. The GPR model not only successfully captures the overall rising and falling trends of CO_2 concentration but also finely characterizes the periodic fluctuation patterns, showing a high degree of consistency with the actual data's variation rhythm.

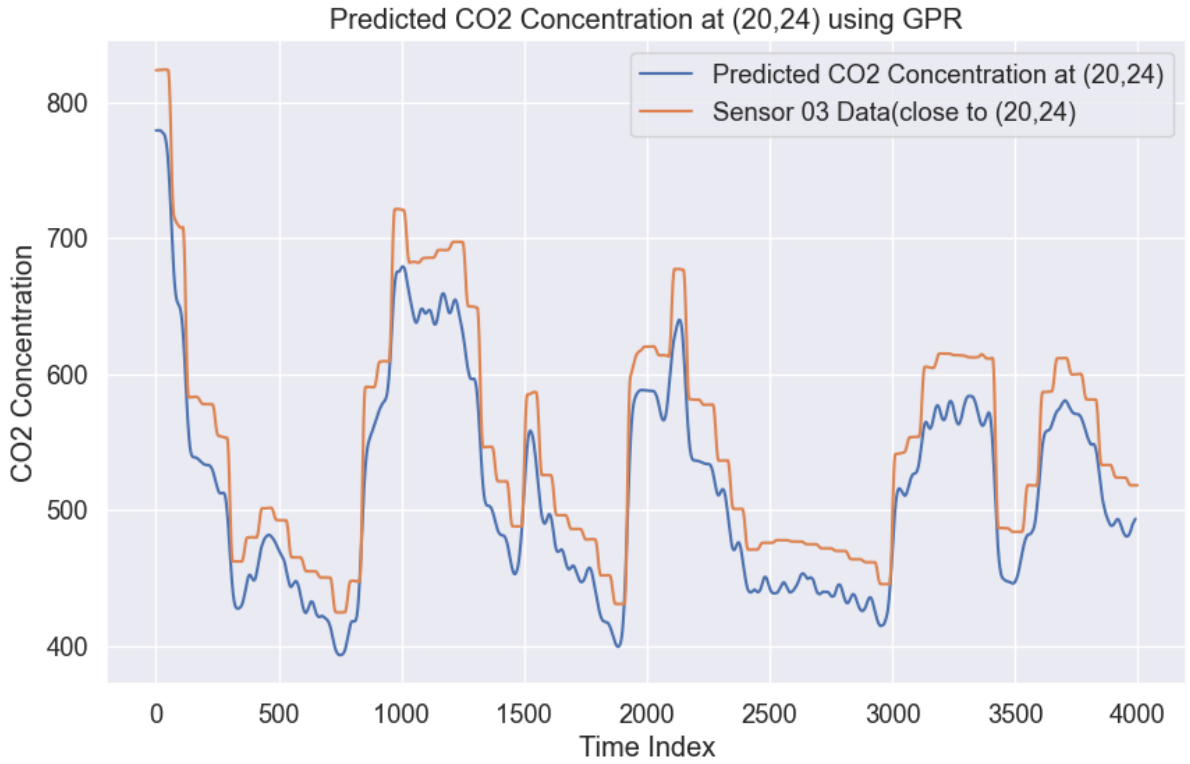


Figure 5.6.1: Time Dimension Prediction

It is noteworthy that the GPR model also demonstrates high precision in the details. For instance, around time index 1000, there is a significant decrease and subsequent rebound in CO_2 concentration, and the GPR model accurately predicts this change.

The predicted CO_2 concentration values closely match the actual data. Additionally, around time indices 2000 and 3000, where two relatively smooth peaks occur, the GPR model accurately forecasts the timing and magnitude of these peaks.

However, it is important to acknowledge that there are minor discrepancies between the predicted results and the actual data. For certain time points, the predicted CO_2 concentration is slightly higher or lower than the actual data, and the fluctuation amplitude of the predicted results shows minor differences compared to the actual data. Such minor differences are entirely normal, as the GPR model cannot capture all factors affecting CO_2 concentration, such as variations in environmental temperature, humidity, pressure, and inherent sensor measurement errors.

In summary, figure 5.6.1 effectively demonstrates the GPR model's exceptional ability in predicting CO_2 concentration. The model accurately forecasts the changes in CO_2 concentration trends and periodic fluctuations and also exhibits high precision in the details. This performance indicates the GPR model's promising application prospects in environmental monitoring and air quality forecasting.

As a comparison, I used the weighted average (WA) model for fusion prediction, the result is shown in figure 5.6.2. But the result is that the RMSE of the GPR model is 35, while the RMSE of the weighted average model is 32, which indicates that the GPR model in this dataset is similarly effective to the WA model, or even slightly less effective than the wa model. But I think the GPR model still has some advantages over the WA model.

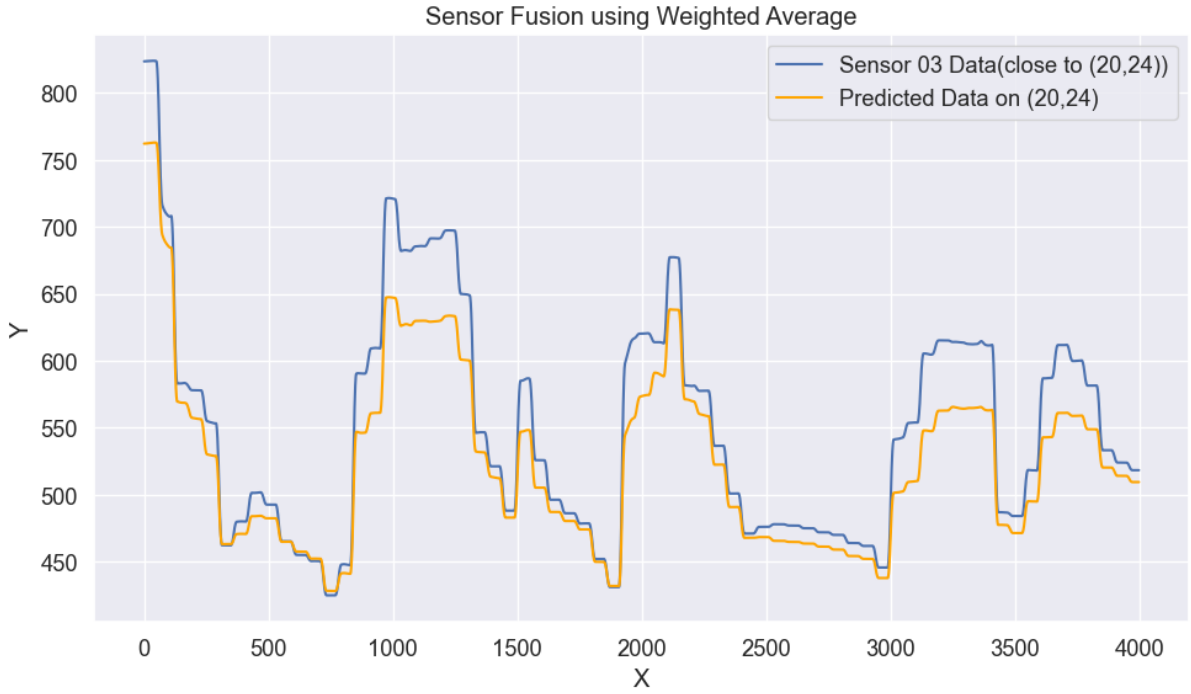


Figure 5.6.2: Time Dimension Prediction

To explain why the GPR model might be superior to the WA model despite the slightly higher RMSE (34 for GPR vs. 32 for WA), we can analyze the data details and focus on aspects like data fluctuations and model stability:

1. Volatility Analysis
The GPR model's predicted curve (blue line) shows smoother fluctuations, especially in sections with rapid increases or decreases. This smooth characteristic indicates that the GPR model is better at handling noise and outliers in the time series data, resulting in more robust predictions.

In contrast, the WA model's predicted curve (yellow line) displays larger volatility, with frequent up-and-down movements in certain time periods. This excessive fluctuation might be because the WA model heavily weights certain sensors, causing it to overreact to noise from these sensors. This can lead to less stable predictions in real-world applications, making the WA model more sensitive to noise.

Detail Capture:
By utilizing Gaussian processes, the GPR model is adept at capturing significant details in the time series, especially during rapid changes. For example, in multiple peaks and valleys within the graph, the GPR model tracks these changes more accurately without introducing unnecessary fluctuations. This capability makes the GPR model provide more accurate and trustworthy predictions during sudden changes.

The WA model, on the other hand, shows an excessive sensitivity at certain time points, resulting in amplified unnecessary details or noise. This is evident in the slight fluctuations during plateau phases (where data remains relatively stable), which could be due to the model overreacting to noise, leading to exaggerated predictions.

2. Stability and Robustness
The GPR model, operating within a Bayesian framework, offers more stable predictions in the presence of data noise. The smooth fluctuations not only enhance prediction accuracy but also improve the model's ability to generalize when faced with unseen data. Stability is crucial in many real-world applications, particularly when continuous long-term predictions are required.

The WA model's higher volatility suggests that it might strongly react to any small changes in input data, which can be a disadvantage in some applications. For example, in environmental monitoring, an overly sensitive model might falsely report anomalies, reducing the reliability of predictions.

3. Error Analysis in Practical Applications
Although the WA model has a slightly lower overall RMSE, its greater volatility might result in significant errors during critical moments. For example, in environmental data analysis, sudden environmental changes could have a major impact on decision-making. In such cases, the smooth characteristics and stability of the GPR model might be more reliable. While its error is slightly higher, it is easier to interpret and manage.

By focusing on these detailed analyses, we can better illustrate why the GPR model, despite having a slightly higher RMSE, provides more stable and reliable predictions in practical applications, making it overall superior to the more volatile WA model.

5.6.2 Space Dimension Prediction

Figure 5.6.3 and figure 5.6.4 compares two 3D plots representing the spatial distribution of CO_2 concentration across a given area. These plots were generated at different time points or under varying conditions. By examining the differences in peak concentrations, distribution patterns, and overall trends, we can gain insights into the dynamics of CO_2 emissions and their environmental impact over time.

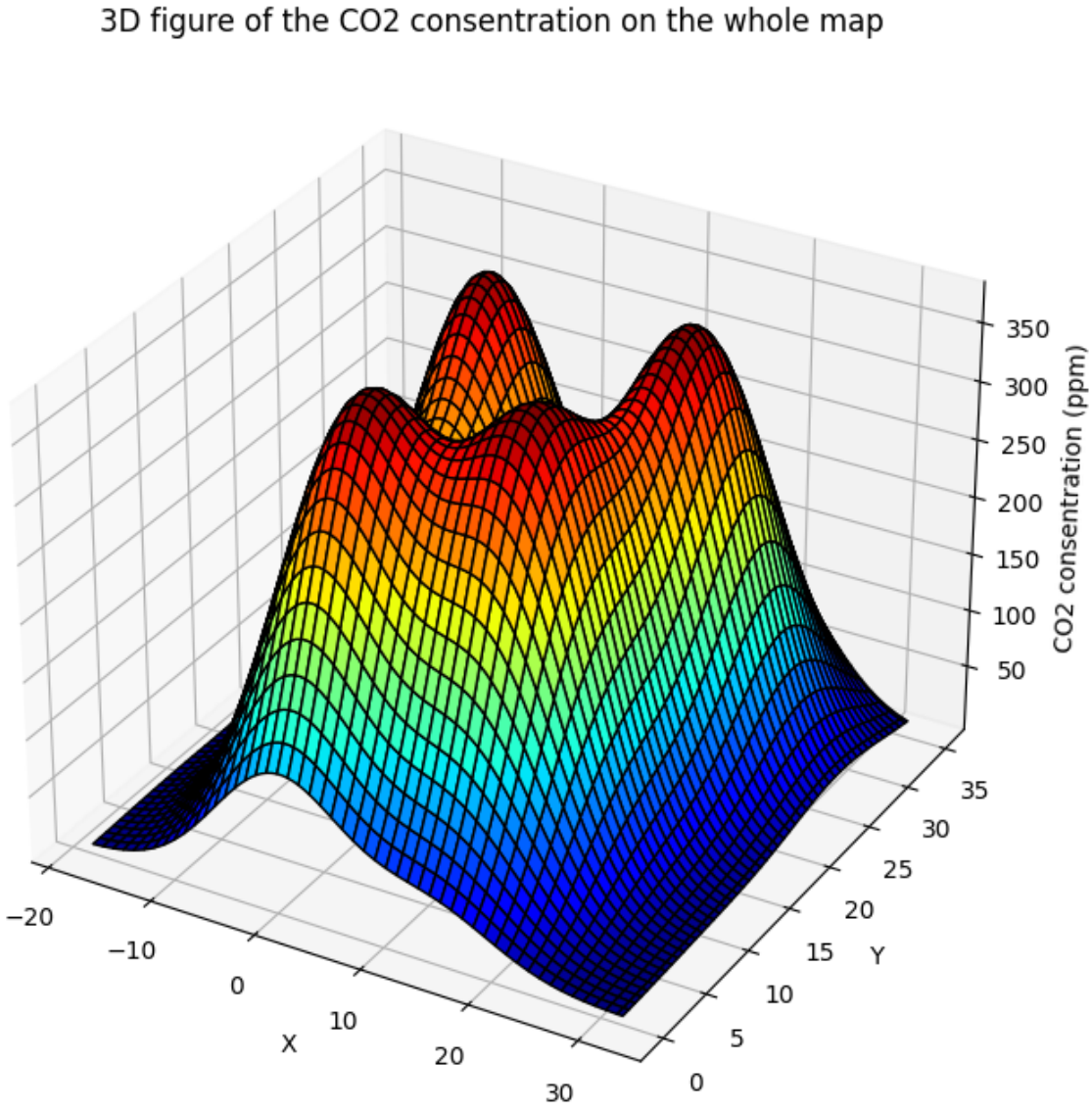


Figure 5.6.3: CO_2 Concentration at 10:35 AM

Firstly, I analyze the CO_2 concentration at the morning time of 10:35 AM, which is shown in figure 5.6.3.

1. **Concentration Peaks**: At 10:35 AM, the visual representation reveals several distinct CO_2 concentration peaks, with the highest values approaching 350 ppm. These peaks are primarily located between X-axis 10 to 20 and Y-axis 15 to 25.

2. **Spatial Distribution**: The elevated concentration areas are predominantly

centralized within the image. This distribution may correspond to specific zones within the building, such as classrooms or meeting spaces, where occupancy has recently commenced. At this hour, these areas might be experiencing initial usage, wherein the ventilation system has yet to fully mitigate the CO_2 increase, resulting in localized accumulation.

3. **Concentration Gradient**: The regions surrounding these high-concentration zones exhibit steep CO_2 gradients, indicative of an uneven spatial distribution. This disparity could be attributed to variations in ventilation effectiveness or differing levels of occupancy across various rooms.

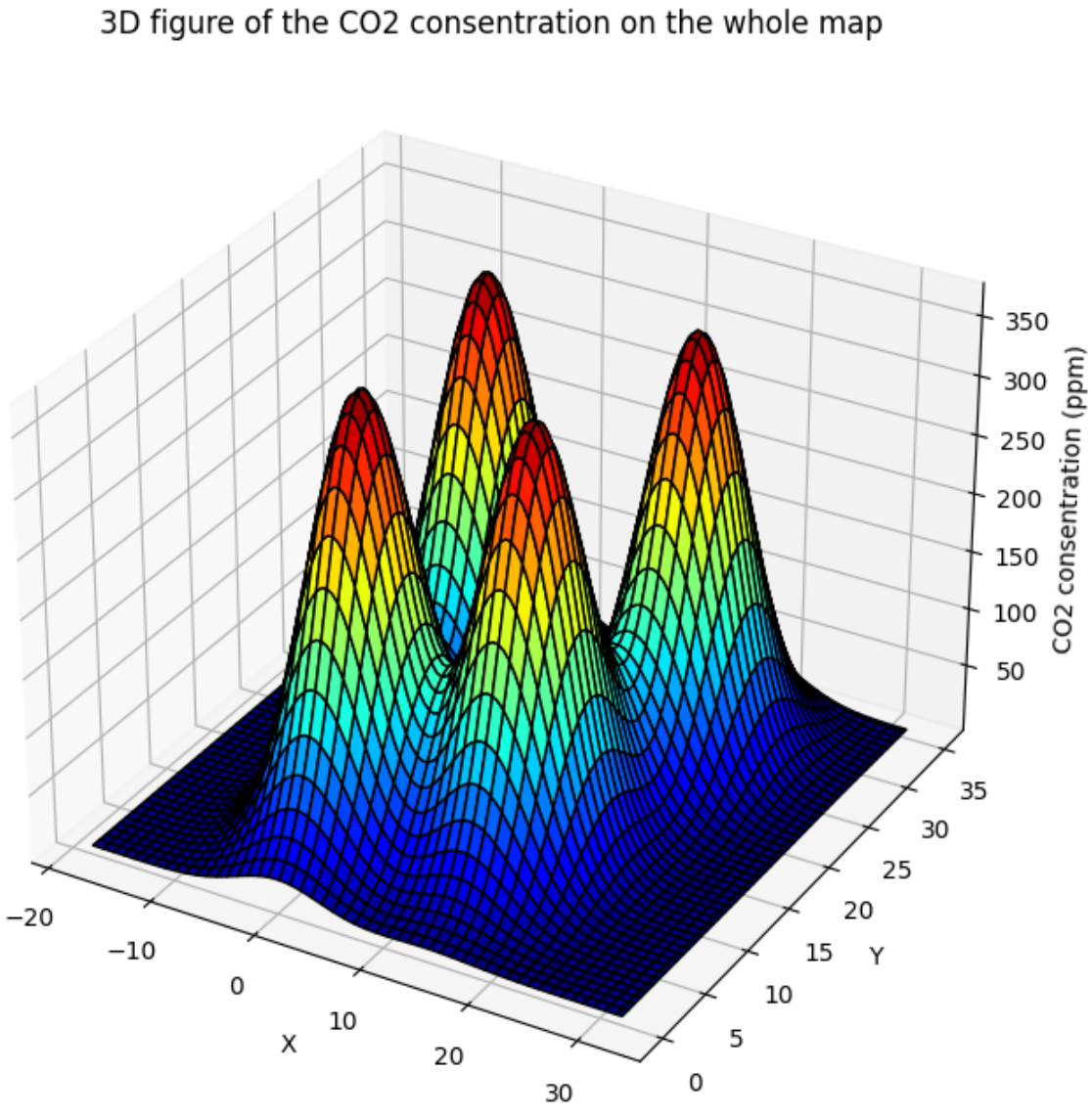


Figure 5.6.4: CO_2 Concentration at 6:30 PM

Figure 5.6.4 (6:30 PM)

Subsequently, attention is directed towards the CO_2 concentration later in the day, specifically at 6:30 PM, which is shown in figure 5.6.4.

1. **Concentration Peaks**: By 6:30 PM, there is a noticeable escalation in overall CO_2 concentration, with peaks reaching approximately 400 ppm. The areas of elevated concentration have broadened, and additional peaks have emerged, particularly between X-axis 20 to 30 and Y-axis 15 to 25.

2. **Spatial Distribution**: In the afternoon, the zones of high concentration become more widespread, suggesting these areas have been subject to prolonged usage, such as continuous classes or extended activities. This also implies that the ventilation system may be inadequate in effectively reducing CO_2 levels over sustained periods.

3. **Concentration Gradient**: The gradient of CO_2 concentration becomes more pronounced, indicating an accumulation of CO_2 in these high-traffic areas. This situation typically arises when occupants remain in a confined space for an extended duration, impeding the dispersal of CO_2 and leading to a significant increase in concentration.

To gain a comprehensive understanding, it is essential to conduct a comparative analysis of the CO_2 concentrations at these two distinct time points.

1. **Impact of Occupancy on CO_2 Levels**:

- **10:35 AM**: At this juncture, classes may have only recently commenced, with students and faculty members entering the classrooms, resulting in an initial rise in CO_2 concentration in specific areas. Nevertheless, as the morning session may not be in full swing, certain regions exhibit lower concentrations, indicating a relatively dispersed occupancy pattern.

- **6:30 PM**: In contrast, by the late afternoon, the academic and extracurricular activities of the day are nearing conclusion. Consequently, classrooms and other activity areas have likely been utilized multiple times, leading to elevated localized CO_2 concentrations. Insufficient ventilation exacerbates the prominence of these high-concentration areas.

2. **Impact of Ventilation**: - The comparative analysis between morning and afternoon reveals a significant increase in CO_2 concentration within particular areas as the day progresses. This observation suggests that the ventilation system may be inadequate, particularly in frequently utilized spaces, leading to a more pronounced accumulation of CO_2 in the afternoon.

3. **Regional Differences**: - There is a notable overlap in the regions of high concentration between the two time points, indicating that these may be core areas of use within the building, such as classrooms, laboratories, or meeting rooms. These zones are characterized by high occupancy and frequent activity, making them more susceptible to CO_2 accumulation.

In conclusion, the observed fluctuations in CO_2 concentration within this educational building underscore the significant impact of occupancy patterns and ventilation efficiency. The morning image at 10:35 AM highlights the initial CO_2 buildup following the commencement of activities, whereas the afternoon image at 6:30 PM illustrates the cumulative effect of prolonged occupancy. To enhance indoor air quality, it is imperative to improve ventilation, particularly in areas that experience high occupancy

and extended usage.

Chapter 6

Conclusions

This research focuses on addressing key challenges in calibrating, short-term forecasting, and sensor fusion of low-cost CO_2 sensors in environmental monitoring. By proposing and validating a series of advanced methodologies, the research contributes to enhancing the accuracy, reliability, and scalability of these sensors. Specifically, the study employs the Deep Calibration Method (DeepCM) to address the non-linear data relationships of sensors, Long Short-Term Memory networks (LSTM) for short-term forecasting, and Gaussian Process Regression (GPR) for sensor data fusion. The application of these methods demonstrates superior performance in various experimental scenarios, providing significant support for the advancement of low-cost sensor technologies.

6.1 Positive Impacts

1. **Improved Calibration Accuracy**: The introduction of the DeepCM model significantly enhances the calibration accuracy of low-cost sensors, enabling them to provide more reliable data in practical applications. Specifically, DeepCM effectively corrects systemic errors and non-linear responses of the sensors through complex mappings provided by deep neural networks.

2. **Effective Short Term Forecasting**: The LSTM model performs exceptionally well in short-term CO_2 concentration forecasting by capturing the temporal dependencies in the data. This model learns both long-term and short-term dependencies from past data, resulting in more accurate predictions and reduced forecast errors.

3. **Enhanced Data Fusion**: Gaussian Process Regression (GPR) shows excellent performance in handling complex noise characteristics in multi-sensor data fusion, significantly improving the accuracy of long-term CO_2 concentration predictions. GPR enhances the reliability and accuracy of fused data by modeling uncertainties and effectively estimating noise.

6.2 Drawbacks

1. **Model Complexity**: The complexity of deep learning models such as DeepCM and LSTM increases computational costs, which may be unsuitable for resource-constrained environments. Specifically, the computational burden and time delay introduced by these models could limit their use in low-resource settings.
2. **High Data Requirements**: The success of these models relies on a large amount of high-quality data, which low-cost sensors may not provide. Insufficient data volume and quality could impact model performance, leading to suboptimal prediction accuracy and stability.
3. **External Factors Impact**: The current models' ability to handle external disturbances (e.g., weather changes, human activities) is not yet fully optimized. This limitation may affect the model's performance and applicability in certain environmental conditions.

6.3 Research Evaluation

The proposed methods, including DeepCM, LSTM, and GPR, have shown outstanding performance in various experimental scenarios, particularly in enhancing the calibration accuracy of low-cost sensors, the accuracy of short-term forecasting, and the effectiveness of data fusion. However, limitations such as model complexity, data requirements, and handling of external disturbances impact the practical application range of these results.

6.4 Future Work

1. **Model Optimization**: Further work focus on the optimization of DeepCM and LSTM models is needed to ensure effective performance in environments with limited computational resources. This may involve exploring more efficient algorithms and simplified network architectures, such as model compression and lightweight designs, to reduce computational demands.
2. **Multi-source Data Fusion**: Future research should explore additional sensor data fusion methods to handle more complex environmental data. By integrating various types of sensor data, the robustness and adaptability of the models can be enhanced.
3. **Consideration of External Factors**: Incorporating mechanisms to address external factors such as weather changes into the models is necessary to improve prediction accuracy and applicability in real-world environments. Enhancing the model's adaptability to external disturbances will be crucial.
4. **Real-time Application Expansion**: Applying the research findings to real-time monitoring systems and validating their effectiveness in practical scenarios is essential. This will not only assess the real-world performance of the models

but also help identify potential issues and make further improvements to enhance practical applicability.

Bibliography

[1] Kumar, Prashant, Morawska, Lidia, Martani, Claudio, Biskos, George, Neophytou, Marina, Di Sabatino, Silvana, Bell, Margaret, Norford, Leslie, and Britter, Rex. "The rise of low-cost sensing for managing air pollution in cities". In: *Environment international* 75 (2015), pp. 199–205.

[2] Poddar, Shashi, Kumar, Vipan, and Kumar, Amod. "A comprehensive overview of inertial sensor calibration techniques". In: *Journal of Dynamic Systems, Measurement, and Control* 139.1 (2017), p. 011006.

[3] Hasenfratz, David, Saukh, Olga, and Thiele, Lothar. "On-the-fly calibration of low-cost gas sensors". In: *European Conference on Wireless Sensor Networks*. Springer. 2012, pp. 228–244.

[4] Mohanty, Sachi Nandan, Lydia, E Laxmi, Elhoseny, Mohamed, Al Otaibi, Majid M Gethami, and Shankar, K. "Deep learning with LSTM based distributed data mining model for energy efficient wireless sensor networks". In: *Physical Communication* 40 (2020), p. 101097.

[5] Yeong, De Jong, Velasco-Hernandez, Gustavo, Barry, John, and Walsh, Joseph. "Sensor and sensor fusion technology in autonomous vehicles: A review". In: *Sensors* 21.6 (2021), p. 2140.

[6] Bossche, Michael van den, Rose, Nathan Tyler, and De Wekker, Stephan Franz Joseph. "Potential of a low-cost gas sensor for atmospheric methane monitoring". In: *Sensors and Actuators B: Chemical* 238 (2017), pp. 501–509.

[7] Sun, Li, Westerdahl, Dane, and Ning, Zhi. "Development and evaluation of a novel and cost-effective approach for low-cost NO2 sensor drift correction". In: *Sensors* 17.8 (2017), p. 1916.

[8] Baron, Ronan and Saffell, John. "Amperometric gas sensors as a low cost emerging technology platform for air quality monitoring applications: A review". In: *ACS sensors* 2.11 (2017), pp. 1553–1566.

[9] Yu, Haomin, Li, Qingyong, Wang, Rao, Chen, Zechuan, Zhang, Yingjun, Geng, Yangli-ao, Zhang, Ling, Cui, Houxin, and Zhang, Ke. "A deep calibration method for low-cost air monitoring sensors with multilevel sequence modeling". In: *IEEE Transactions on Instrumentation and Measurement* 69.9 (2020), pp. 7167–7179.

[10] Malhotra, Pankaj, Ramakrishnan, Anusha, Anand, Gaurangi, Vig, Lovekesh, Agarwal, Puneet, and Shroff, Gautam. "LSTM-based encoder-decoder for multi-sensor anomaly detection". In: *arXiv preprint arXiv:1607.00148* (2016).

[11] Yaro, Abdulmalik Shehu, Maly, Filip, and Prazak, Pavel. “Outlier Detection in Time-Series Receive Signal Strength Observation Using Z-Score Method with S n Scale Estimator for Indoor Localization”. In: *Applied Sciences* 13.6 (2023), p. 3900.

[12] Adikaram, KK Lasantha Britto, Hussein, Mohamed A, Effenberger, Mathias, and Becker, Thomas. “Data transformation technique to improve the outlier detection power of Grubbs’ test for data expected to follow linear relation”. In: *Journal of applied mathematics* 2015.1 (2015), p. 708948.

[13] Adikaram, KK Lasantha Britto, Hussein, Mohamed A, Effenberger, Mathias, and Becker, Thomas. “Data transformation technique to improve the outlier detection power of Grubbs’ test for data expected to follow linear relation”. In: *Journal of applied mathematics* 2015.1 (2015), p. 708948.

[14] Breunig, Markus M, Kriegel, Hans-Peter, Ng, Raymond T, and Sander, Jörg. “LOF: identifying density-based local outliers”. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 93–104.

[15] Cheng, Zhangyu, Zou, Chengming, and Dong, Jianwei. “Outlier detection using isolation forest and local outlier factor”. In: *Proceedings of the conference on research in adaptive and convergent systems*. 2019, pp. 161–168.

[16] Thongkam, Jaree, Xu, Guandong, Zhang, Yanchun, and Huang, Fuchun. “Support vector machine for outlier detection in breast cancer survivability prediction”. In: *Advanced Web and Network Technologies, and Applications: APWeb 2008 International Workshops: BIDM, IWHDM, and DeWeb Shenyang, China, April 26-28, 2008. Revised Selected Papers 10*. Springer. 2008, pp. 99–109.

[17] Jiang, Sheng-yi and An, Qing-bo. “Clustering-based outlier detection method”. In: *2008 Fifth international conference on fuzzy systems and knowledge discovery*. Vol. 2. IEEE. 2008, pp. 429–433.

[18] Khan, Kamran, Rehman, Saif Ur, Aziz, Kamran, Fong, Simon, and Sarasvady, Sababady. “DBSCAN: Past, present and future”. In: *The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014)*. IEEE. 2014, pp. 232–238.

[19] Yu, Yufeng, Zhu, Yuelong, Li, Shijin, and Wan, Dingsheng. “Time series outlier detection based on sliding window prediction”. In: *Mathematical problems in Engineering* 2014.1 (2014), p. 879736.

[20] Zellner, Ludwig, Richter, Florian, Sontheim, Janina, Maldonado, Andrea, and Seidl, Thomas. “Concept drift detection on streaming data with dynamic outlier aggregation”. In: *Process Mining Workshops: ICPM 2020 International Workshops, Padua, Italy, October 5–8, 2020, Revised Selected Papers 2*. Springer. 2021, pp. 206–217.

[21] Abbas, Nasir, Abujiya, Mu’azu Ramat, Riaz, Muhammad, and Mahmood, Tahir. “Cumulative sum chart modeled under the presence of outliers”. In: *Mathematics* 8.2 (2020), p. 269.

[22] Chen, Jinghui, Sathe, Saket, Aggarwal, Charu, and Turaga, Deepak. “Outlier detection with autoencoder ensembles”. In: *Proceedings of the 2017 SIAM international conference on data mining*. SIAM. 2017, pp. 90–98.

[23] Chen, Jinghui, Sathe, Saket, Aggarwal, Charu, and Turaga, Deepak. “Outlier detection with autoencoder ensembles”. In: *Proceedings of the 2017 SIAM international conference on data mining*. SIAM. 2017, pp. 90–98.

[24] Nguyen, Nam H, Nguyen, Huy X, Le, Thuan TB, and Vu, Chinh D. “Evaluating low-cost commercially available sensors for air quality monitoring and application of sensor calibration methods for improving accuracy”. In: (2021).

[25] Cohn, Gabe, Gupta, Sidhant, Froehlich, Jon, Larson, Eric, and Patel, Shwetak N. “GasSense: Appliance-level, single-point sensing of gas activity in the home”. In: *Pervasive Computing: 8th International Conference, Pervasive 2010, Helsinki, Finland, May 17-20, 2010. Proceedings 8*. Springer. 2010, pp. 265–282.

[26] Hou, Yuanbin, Fan, Rong, Chen, Xizi, and Dong, Tiantian. “The compensation algorithm based on Bayes theory of multi point measurement fusion for methane concentration”. In: *2016 International Symposium on Computer, Consumer and Control (IS3C)*. IEEE. 2016, pp. 536–540.

[27] Björkman, Mats. “Knowledge, calibration, and resolution: A linear model”. In: *Organizational Behavior and Human Decision Processes* 51.1 (1992), pp. 1–21.

[28] McCrackin, FL and Chang, SS. “Simple calibration procedures for platinum resistance thermometers from 2.5 to 14 K”. In: *Review of Scientific Instruments* 46.5 (1975), pp. 550–553.

[29] Wang, Zhongyu, Li, Qiang, Wang, Zhuoran, and Yan, Hu. “Novel method for processing the dynamic calibration signal of pressure sensor”. In: *Sensors* 15.7 (2015), pp. 17748–17766.

[30] Xie, Wenjun and Bai, Peng. “A pressure sensor calibration model based on support vector machine”. In: *2012 24th Chinese Control and Decision Conference (CCDC)*. IEEE. 2012, pp. 3239–3242.

[31] Zimmerman, Naomi, Presto, Albert A, Kumar, Sriniwasa PN, Gu, Jason, Hauryliuk, Aliaksei, Robinson, Ellis S, Robinson, Allen L, and Subramanian, Ramachandran. “A machine learning calibration model using random forests to improve sensor performance for lower-cost air quality monitoring”. In: *Atmospheric Measurement Techniques* 11.1 (2018), pp. 291–313.

[32] Zhao, Ganning, Hu, Jiesi, You, Suya, and Kuo, C-C Jay. “Calibdnn: multimodal sensor calibration for perception using deep neural networks”. In: *Signal Processing, Sensor/Information Fusion, and Target Recognition XXX*. Vol. 11756. SPIE. 2021, pp. 324–335.

[33] Ali, Sharafat, Alam, Fakhrul, Arif, Khalid Mahmood, and Potgieter, Johan. “Low-cost CO sensor calibration using one dimensional convolutional neural network”. In: *Sensors* 23.2 (2023), p. 854.

[34] Yu, Haomin, Li, Qingyong, Wang, Rao, Chen, Zechuan, Zhang, Yingjun, Geng, Yangli-ao, Zhang, Ling, Cui, Houxin, and Zhang, Ke. “A deep calibration method for low-cost air monitoring sensors with multilevel sequence modeling”. In: *IEEE Transactions on Instrumentation and Measurement* 69.9 (2020), pp. 7167–7179.

[35] Ostertagová, Eva and Ostertag, Oskar. “The simple exponential smoothing model”. In: *The 4th International Conference on modelling of mechanical and mechatronic systems, Technical University of Košice, Slovak Republic, Proceedings of Conference*. 2011, pp. 380–384.

[36] LaViola, Joseph J. “Double exponential smoothing: an alternative to Kalman filter-based predictive tracking”. In: *Proceedings of the workshop on Virtual environments 2003*. 2003, pp. 199–206.

[37] Box, George EP and Pierce, David A. “Distribution of residual autocorrelations in autoregressive-integrated moving average time series models”. In: *Journal of the American statistical Association* 65.332 (1970), pp. 1509–1526.

[38] Arumugam, Vignesh and Natarajan, Vijayalakshmi. “Time Series Modeling and Forecasting Using Autoregressive Integrated Moving Average and Seasonal Autoregressive Integrated Moving Average Models.” In: *Instrumentation, Mesures, Métrologies* 22.4 (2023).

[39] Wan, Renzhuo, Mei, Shuping, Wang, Jun, Liu, Min, and Yang, Fan. “Multivariate temporal convolutional network: A deep neural networks approach for multivariate time series forecasting”. In: *Electronics* 8.8 (2019), p. 876.

[40] Welch, Greg, Bishop, Gary, et al. “An introduction to the Kalman filter”. In: (1995).

[41] Fujii, Keisuke. “Extended kalman filter”. In: *Refernce Manual* 14 (2013), p. 41.

[42] Wan, Eric A and Van Der Merwe, Rudolph. “The unscented Kalman filter”. In: *Kalman filtering and neural networks* (2001), pp. 221–280.

[43] Djuric, Petar M, Kotecha, Jayesh H, Zhang, Jianqui, Huang, Yufei, Ghirmai, Tadesse, Bugallo, Mónica F, and Miguez, Joaquin. “Particle filtering”. In: *IEEE signal processing magazine* 20.5 (2003), pp. 19–38.

[44] Doucet, Arnaud, De Freitas, Nando, and Gordon, Neil. “An introduction to sequential Monte Carlo methods”. In: *Sequential Monte Carlo methods in practice* (2001), pp. 3–14.

[45] Nakamura, Eduardo F, Loureiro, Antonio AF, and Frery, Alejandro C. “Information fusion for wireless sensor networks: Methods, models, and classifications”. In: *ACM Computing Surveys (CSUR)* 39.3 (2007), 9–es.

[46] Liu, Jia, Li, Tianrui, Xie, Peng, Du, Shengdong, Teng, Fei, and Yang, Xin. “Urban big data fusion based on deep learning: An overview”. In: *Information Fusion* 53 (2020), pp. 123–133.

[47] Münzner, Sebastian, Schmidt, Philip, Reiss, Attila, Hanselmann, Michael, Stiefelhagen, Rainer, and Dürichen, Robert. “CNN-based sensor fusion techniques for multimodal human activity recognition”. In: *Proceedings of the 2017 ACM international symposium on wearable computers*. 2017, pp. 158–165.

[48] Saleh, Khaled, Hossny, Mohammed, and Nahavandi, Saeid. “Driving behavior classification based on sensor data fusion using LSTM recurrent neural networks”. In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2017, pp. 1–6.

[49] Shumway, Robert H, Stoffer, David S, Shumway, Robert H, and Stoffer, David S. “ARIMA models”. In: *Time series analysis and its applications: with R examples* (2017), pp. 75–163.

[50] Demir, Volkan, Zontul, Metin, and Yelmen, Ilkay. “Drug sales prediction with ACF and PACF supported ARIMA method”. In: *2020 5th International Conference on Computer Science and Engineering (UBMK)*. IEEE. 2020, pp. 243–247.

[51] Wang, Yuanyuan, Wang, Jianzhou, Zhao, Ge, and Dong, Yao. “Application of residual modification approach in seasonal ARIMA for electricity demand forecasting: A case study of China”. In: *Energy Policy* 48 (2012), pp. 284–294.

[52] Arumugam, Paramasivan and Saranya, R. “Outlier detection and missing value in seasonal ARIMA model using rainfall data”. In: *Materials Today: Proceedings* 5.1 (2018), pp. 1791–1799.

[53] Obikee, Adaku C, Ebuh, Godday U, Obiora-Ilouno, Happiness O, et al. “Comparison of outlier techniques based on simulated data”. In: *Open Journal of Statistics* 4.07 (2014), p. 536.

[54] Grossberg, Stephen. “Recurrent neural networks”. In: *Scholarpedia* 8.2 (2013), p. 1888.

[55] Li, Xi-Lin. “Recurrent neural network training with preconditioned stochastic gradient descent”. In: *arXiv preprint arXiv:1606.04449* (2016).

[56] Chandriah, Kiran Kumar and Naraganahalli, Raghavendra V. “RNN/LSTM with modified Adam optimizer in deep learning approach for automobile spare parts demand forecasting”. In: *Multimedia Tools and Applications* 80.17 (2021), pp. 26145–26159.

[57] Yao, Kaisheng, Cohn, Trevor, Vylomova, Katerina, Duh, Kevin, and Dyer, Chris. “Depth-gated LSTM”. In: *arXiv preprint arXiv:1508.03790* (2015).

[58] Gunawardena, Jeremy. “Min-max functions”. In: *Discrete Event Dynamic Systems* 4 (1994), pp. 377–407.

[59] Graves, Alex and Graves, Alex. “Long short-term memory”. In: *Supervised sequence labelling with recurrent neural networks* (2012), pp. 37–45.

[60] Wang, Zhumei, Su, Xing, and Ding, Zhiming. “Long-term traffic prediction based on lstm encoder-decoder architecture”. In: *IEEE Transactions on Intelligent Transportation Systems* 22.10 (2020), pp. 6561–6571.

[61] Du, Shengdong, Li, Tianrui, Yang, Yan, and Horng, Shi-Jinn. “Multivariate time series forecasting via attention-based encoder–decoder framework”. In: *Neurocomputing* 388 (2020), pp. 269–279.

[62] Saurabh, Nikitha. “LSTM-RNN model to predict future stock prices using an efficient optimizer”. In: *International Research Journal of Engineering and Technology (IRJET)* 7.11 (2020), p. 672.

[63] Zhang, Bo, Zou, Guojian, Qin, Dongming, Lu, Yunjie, Jin, Yupeng, and Wang, Hui. “A novel Encoder-Decoder model based on read-first LSTM for air pollutant prediction”. In: *Science of The Total Environment* 765 (2021), p. 144507.

[64] Schulz, Eric, Speekenbrink, Maarten, and Krause, Andreas. “A tutorial on Gaussian process regression: Modelling, exploring, and exploiting functions”. In: *Journal of mathematical psychology* 85 (2018), pp. 1–16.

[65] Nath, Bai Kunth and Bhuiyan, Alauddin. “A Geometrical Feature Based Sensor Fusion Model of GPR and IR for the Detection and Classification of Anti-Personnel Mines”. In: *Seventh International Conference on Intelligent Systems Design and Applications (ISDA 2007)*. 2007, pp. 849–856. doi: 10.1109/ISDA.2007.21.

[66] Annan, AP. “GPR—History, trends, and future developments”. In: *Subsurface sensing technologies and applications* 3.4 (2002), pp. 253–270.

[67] Blu, Thierry, Thévenaz, Philippe, and Unser, Michael. “Linear interpolation revitalized”. In: *IEEE Transactions on Image Processing* 13.5 (2004), pp. 710–719.

[68] Fan, Wenkai, Han, Feiyang, and Wei, Yimin. “Regularized TLS-EM for estimating missing data”. In: *Computational and Applied Mathematics* 43.1 (2024), p. 43.

[69] Schubert, Erich, Sander, Jörg, Ester, Martin, Kriegel, Hans Peter, and Xu, Xiaowei. “DBSCAN revisited, revisited: why and how you should (still) use DBSCAN”. In: *ACM Transactions on Database Systems (TODS)* 42.3 (2017), pp. 1–21.

[70] Çelik, Mete, Dadaşer-Çelik, Filiz, and Dokuz, Ahmet Şakir. “Anomaly detection in temperature data using DBSCAN algorithm”. In: *2011 international symposium on innovations in intelligent systems and applications*. IEEE. 2011, pp. 91–95.

[71] Yu, Haomin, Li, Qingyong, Wang, Rao, Chen, Zechuan, Zhang, Yingjun, Geng, Yangli-ao, Zhang, Ling, Cui, Houxin, and Zhang, Ke. “A deep calibration method for low-cost air monitoring sensors with multilevel sequence modeling”. In: *IEEE Transactions on Instrumentation and Measurement* 69.9 (2020), pp. 7167–7179.

[72] Zhang, Huihui, Li, Shicheng, Chen, Yu, Dai, Jiangyan, and Yi, Yugen. “A Novel Encoder-Decoder Model for Multivariate Time Series Forecasting”. In: *Computational intelligence and neuroscience* 2022.1 (2022), p. 5596676.

[73] Liu, Zitong, Ding, Guoru, Wang, Zheng, Zheng, Shilian, Sun, Jiachen, and Wu, Qihui. “Cooperative topology sensing of wireless networks with distributed sensors”. In: *IEEE Transactions on Cognitive Communications and Networking* 7.2 (2020), pp. 524–540.