

## *Report*

---

# Hurricane Intensity Prediction

Author: Daanyaal Parvaize, Keerti Belmane, Kreetika Mohanta  
Matriculation No.: 7026795, 7026791 7026001  
Course of Study: Business Intelligence and Data Analytics  
  
First examiner: Prof. Dr. Elmar Wings  
Submission date: June 30, 2025

# Contents

<b>Contents</b>	<b>i</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Listings</b>	<b>xi</b>
<b>Acronyms</b>	<b>xi</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Problem Description: Hurricane Wind Speed Prediction Using Time Series Models . . . . .	2
1.1.1 Background . . . . .	2
1.1.2 Problem Statement . . . . .	2
1.1.3 Literature Overview . . . . .	2
1.1.4 Challenges . . . . .	3
1.1.5 Proposed Solution . . . . .	3
1.1.6 Model Selection Guide . . . . .	4
1.1.7 Advantages . . . . .	5
1.1.8 Report Structure . . . . .	5
<b>2 Domain Knowledge</b>	<b>8</b>
2.1 Application . . . . .	8
2.2 Problem . . . . .	8
2.2.1 Complex Natural Factors . . . . .	8
2.2.2 Imperfect Data . . . . .	8
2.2.3 Track vs. Intensity Prediction . . . . .	8
2.2.4 Rapid Intensification Events . . . . .	9
2.2.5 ARIMA Limitations . . . . .	9
2.3 Data Acquisition . . . . .	9
2.4 Data Quantity . . . . .	10
2.5 Data Quality . . . . .	10
2.5.1 Missing Values . . . . .	10
2.5.2 Formatting Issues . . . . .	10
2.6 Data Relevance . . . . .	11
2.7 Outliers . . . . .	11

2.8	Anomalies . . . . .	12
2.9	Summary . . . . .	12
<b>3</b>	<b>Important Modules</b>	<b>13</b>
3.1	Pandas . . . . .	13
3.1.1	Pandas Overview . . . . .	13
3.1.2	Pandas Functionality . . . . .	13
3.1.3	Installing Pandas . . . . .	14
3.1.4	Example: Using Pandas . . . . .	15
3.1.5	Example – Manual . . . . .	16
3.1.6	Example – Code . . . . .	16
3.1.7	Example – Files . . . . .	17
3.1.8	Further Reading . . . . .	17
3.2	Statsmodels . . . . .	17
3.2.1	Introduction . . . . .	17
3.2.2	Description . . . . .	18
3.2.3	Installation . . . . .	18
3.2.4	Example – Description . . . . .	20
3.2.5	Example – Manual . . . . .	20
3.2.6	Example – Code . . . . .	21
3.2.7	Example – Files . . . . .	21
3.2.8	Further Reading . . . . .	22
3.3	TensorFlow . . . . .	22
3.3.1	Introduction . . . . .	22
3.3.2	Description . . . . .	22
3.3.3	Installation . . . . .	23
3.3.4	Example – Description . . . . .	25
3.3.5	Example – Manual . . . . .	25
3.3.6	Example – Code . . . . .	26
3.3.7	Example – Files . . . . .	26
3.3.8	Further Reading . . . . .	27
3.4	NumPy . . . . .	27
3.4.1	Introduction . . . . .	27
3.4.2	Description . . . . .	27
3.4.3	Installation . . . . .	27
3.4.4	Example - Description . . . . .	29
3.4.5	Example - Manual . . . . .	29
3.4.6	Example - Code . . . . .	30
3.4.7	Example - Files . . . . .	30
3.4.8	Further Reading . . . . .	30
3.5	Matplotlib . . . . .	31
3.5.1	Introduction . . . . .	31
3.5.2	Description . . . . .	31
3.5.3	Installation . . . . .	31

---

3.5.4	Example – Description . . . . .	33
3.5.5	Example – Manual . . . . .	33
3.5.6	Example – Code . . . . .	33
3.5.7	Example – Files . . . . .	34
3.5.8	Further Reading . . . . .	35
3.6	Scikit-learn . . . . .	35
3.6.1	Introduction . . . . .	35
3.6.2	Description . . . . .	35
3.6.3	Installation . . . . .	36
3.6.4	Example – Description . . . . .	38
3.6.5	Example – Manual . . . . .	38
3.6.6	Example – Code . . . . .	39
3.6.7	Example – Files . . . . .	39
3.6.8	Further Reading . . . . .	40
<b>4</b>	<b>Data Mining</b>	<b>41</b>
4.1	Stage Data Mining . . . . .	41
4.1.1	ARIMA Model . . . . .	41
4.1.2	LSTM Model . . . . .	42
4.1.3	Implementation Alignment . . . . .	42
4.1.4	Algorithm Description . . . . .	42
4.1.5	Applications . . . . .	43
4.1.6	Relevance . . . . .	44
4.1.7	Hyperparameters . . . . .	44
4.1.8	Data and Computation Requirements . . . . .	45
4.1.9	Input and Output . . . . .	46
4.1.10	Input . . . . .	46
4.1.11	Output . . . . .	47
4.1.12	Example Using Python Code . . . . .	48
4.1.13	Further Readings . . . . .	50
<b>5</b>	<b>Development Environment</b>	<b>51</b>
5.1	Python Version . . . . .	51
5.2	Description . . . . .	51
5.3	Installation . . . . .	51
5.4	Configuration . . . . .	52
5.5	First Steps . . . . .	54
5.6	Hello World Program . . . . .	54
<b>6</b>	<b>Methodolgy</b>	<b>55</b>
6.1	Standards . . . . .	55
6.1.1	CRISP-DM . . . . .	55
6.1.2	KDD Process . . . . .	56
6.1.3	ML Pipeline . . . . .	57

6.1.4	Other Methodologies . . . . .	59
6.1.5	Justification . . . . .	60
<b>7</b>	<b>Documentation developer</b>	<b>61</b>
7.1	Structure . . . . .	61
7.2	Size . . . . .	63
7.3	Format . . . . .	63
7.4	Anomalies . . . . .	63
7.4.1	Important Columns . . . . .	63
7.5	Origin . . . . .	64
<b>8</b>	<b>KDD-Analysis of the Atlantic Storms Dataset (1975–2021)</b>	<b>65</b>
8.1	Origin . . . . .	65
8.2	Features . . . . .	65
8.3	Data Types . . . . .	66
8.4	Quality . . . . .	67
8.4.1	Completeness . . . . .	67
8.4.2	Accuracy . . . . .	67
8.4.3	Consistency . . . . .	67
8.4.4	Missing Values . . . . .	67
8.5	Quantity . . . . .	68
8.6	Fairness and Bias . . . . .	68
8.6.1	Geographic Bias . . . . .	68
8.6.2	Temporal Bias . . . . .	68
8.6.3	Selection Bias . . . . .	69
8.6.4	Synthetic Data Bias . . . . .	69
8.6.5	Normalization Bias . . . . .	69
8.6.6	Fairness Considerations . . . . .	69
8.7	One Database . . . . .	70
8.8	Properties . . . . .	70
8.9	Outliers . . . . .	71
8.9.1	Wind Speed . . . . .	71
8.9.2	Pressure . . . . .	71
8.9.3	Latitude and Longitude . . . . .	71
8.9.4	Storm Size . . . . .	71
8.9.5	Potential Outliers . . . . .	71
8.10	Anomalies . . . . .	71
8.10.1	Data Entry Issues . . . . .	72
8.10.2	Meteorological Anomalies . . . . .	72
8.10.3	Inconsistent Status . . . . .	72
8.10.4	Historical Data Gaps . . . . .	72
8.10.5	Mitigation Through Augmentation . . . . .	73
8.11	Augmentation . . . . .	73
8.11.1	Handling Missing Values . . . . .	73

8.11.2	Data Standardization and Cleaning . . . . .	74
8.11.3	Synthetic Data Generation . . . . .	75
8.11.4	Normalization for Deep Learning . . . . .	75
8.11.5	Sequence Creation for Deep Learning . . . . .	76
8.11.6	Error Handling and Robustness . . . . .	76
8.11.7	Summary of Augmentation . . . . .	77
<b>9</b>	<b>Data Transformation and Data Mining</b>	<b>78</b>
9.1	Data Transformation . . . . .	78
9.1.1	Data Loading and Cleaning . . . . .	78
9.1.2	Data Scaling . . . . .	78
9.1.3	Sequence Creation . . . . .	78
9.2	Data Mining . . . . .	78
9.2.1	ARIMA Model . . . . .	79
9.2.2	LSTM Model . . . . .	79
9.3	Application to the Project . . . . .	79
9.4	Hyperparameters . . . . .	79
9.4.1	ARIMA Hyperparameters . . . . .	79
9.4.2	LSTM Hyperparameters . . . . .	79
9.5	Input . . . . .	80
9.6	Training . . . . .	80
9.6.1	ARIMA Training . . . . .	80
9.6.2	LSTM Training . . . . .	80
9.7	Interpretation . . . . .	80
9.7.1	ARIMA Forecasts . . . . .	80
9.7.2	LSTM Forecasts . . . . .	80
9.8	Output . . . . .	81
9.9	Visualization of Output . . . . .	81
9.9.1	Diagram-Visualization Of Output . . . . .	81
<b>10</b>	<b>Survey Note: Detailed Analysis of Data Transformation and Data Mining Application</b>	<b>83</b>
10.1	Data Transformation Analysis . . . . .	83
10.1.1	Loading and Cleaning . . . . .	83
10.1.2	Scaling and Sequence Creation . . . . .	83
10.2	Data Mining Analysis . . . . .	83
10.2.1	Linear Pattern Extraction via ARIMA . . . . .	83
10.2.2	Non-linear Dynamics with LSTM . . . . .	84
10.3	Application Overview . . . . .	84
<b>11</b>	<b>Developer Documentation</b>	<b>85</b>
11.1	Introduction . . . . .	85
11.2	Development Idea and Objectives . . . . .	85

11.3	System Architecture and Flowcharts . . . . .	86
11.3.1	Overall Pipeline Flowchart . . . . .	88
11.3.2	ARIMA and LSTM Training Flowcharts with Explanations . . . . .	90
11.4	Notation and Terminology . . . . .	94
11.5	Completeness of the Implementation . . . . .	95
11.6	Machine Learning Pipeline Details . . . . .	96
11.6.1	ML Pipeline Overview-Diagram . . . . .	99
11.7	Program Readability and Coding Practices . . . . .	100
11.7.1	Project Structure and Modularization Explained . . . . .	100
11.7.2	Why Modularization Matters (Simple Explanation) . . . . .	100
11.7.3	Visualizing the Project Structure . . . . .	101
11.7.4	Parameter Handling . . . . .	101
11.7.5	Error Handling and Messaging . . . . .	103
11.8	Future Work and Improvements . . . . .	104
11.8.1	Project File Structure: <code>hurricane_predictor_ready</code> . . . . .	105
<b>12</b>	<b>Development to Deployment</b>	<b>107</b>
12.1	Overview . . . . .	107
12.1.1	Tools Used in Development to Deployment . . . . .	107
12.1.2	Data Structure Files . . . . .	108
12.1.3	Data Flow and File Structure . . . . .	108
12.1.4	TikZ Diagram of Workflow . . . . .	109
12.1.5	Supported File Formats . . . . .	109
12.1.6	Python Code Snippets . . . . .	110
<b>13</b>	<b>Evaluation and Reflection</b>	<b>111</b>
13.1	Evaluation . . . . .	111
13.1.1	Validation . . . . .	112
13.1.2	Conclusion . . . . .	112
<b>14</b>	<b>Monitoring and Robustness</b>	<b>114</b>
14.0.1	Idea . . . . .	114
14.0.2	Alignment with PANKTI . . . . .	114
14.0.3	Monitoring Plan . . . . .	115
14.0.4	Getting New Data . . . . .	116
14.0.5	Evaluation Checks . . . . .	116
14.0.6	Code: Functions . . . . .	117
14.0.7	Code: Test . . . . .	119
14.0.8	Privacy . . . . .	121
14.0.9	Robustness Features . . . . .	121
14.0.10	End-to-End User Process . . . . .	121

<b>15 Deployment</b>	<b>122</b>
15.1 Application Description . . . . .	122
15.1.1 Structure . . . . .	122
15.1.2 Idea . . . . .	123
15.1.3 Application Flow Chart . . . . .	124
15.1.4 Machine Learning Pipeline (Deployment View) . . .	124
15.1.5 Machine Learning Pipeline (Deployment View) . .	126
15.2 Program Structure and Components . . . . .	127
15.2.1 Program Readability . . . . .	127
15.2.2 Structure and Modules . . . . .	127
15.2.3 Parameter Handling . . . . .	127
15.2.4 Error Handling . . . . .	129
15.2.5 Message Handling . . . . .	129
15.3 Nomenclature in Deployment . . . . .	131
15.3.1 Folder Names . . . . .	131
15.3.2 File Names . . . . .	131
15.3.3 Module and Function Names . . . . .	132
15.3.4 Key Variable Names . . . . .	132
<b>16 Test For Software</b>	<b>133</b>
16.1 Test Procedure . . . . .	133
16.2 Test Items . . . . .	134
16.3 Test Cases and Results . . . . .	134
16.4 Test Data . . . . .	135
16.5 Expected Results . . . . .	136
16.6 Reporting and Logging . . . . .	136
16.7 Roles and Responsibilities . . . . .	136
16.8 Schedule and Frequency . . . . .	137
16.9 Automation Guide . . . . .	137
16.9.1 Purpose of Automation . . . . .	137
16.9.2 Prerequisites . . . . .	137
16.9.3 Setup Instructions . . . . .	137
16.9.4 Running the Automated Tests . . . . .	138
16.9.5 Manual GUI Testing . . . . .	139
16.9.6 Recommended Continuous Integration . . . . .	139
16.9.7 Automation Flowchart . . . . .	139
<b>17 Technical Foundations</b>	<b>141</b>
17.1 ARIMA Model Formulation . . . . .	141
17.2 LSTM Network Formulation . . . . .	141
17.3 Forecasting Pipeline . . . . .	142
17.4 Evaluation Metrics . . . . .	142
17.4.1 Root Mean Square Error (RMSE) . . . . .	142
17.4.2 Mean Absolute Error (MAE) . . . . .	142



17.4.3	Mean Absolute Percentage Error (MAPE) . . . . .	143
17.4.4	Coefficient of Determination ( $R^2$ Score) . . . . .	143
17.5	Deployment Error Margin . . . . .	143
17.5.1	Handling Missing Data: Forward-Backward Interpolation . . . . .	143
<b>18</b>	<b>Bill of Materials</b>	<b>144</b>
18.1	Material List and Hardware Bill of Materials . . . . .	144
18.2	Software Requirements and Python Libraries . . . . .	146
18.3	requirements.txt . . . . .	147
18.4	Explanation of Key Libraries . . . . .	147
<b>19</b>	<b>Given documentation</b>	<b>148</b>
19.1	Use of Best Practice Document . . . . .	148
19.2	Same Notation . . . . .	148
19.3	Supplements . . . . .	148
<b>20</b>	<b>Improvements</b>	<b>149</b>
20.1	Minor Changes . . . . .	149
20.1.1	List of Minor Changes . . . . .	149
20.2	Major Improvements . . . . .	149
20.2.1	Documentation of the Major Improvements . . . . .	149
	<b>Bibliography</b>	<b>150</b>

# List of Figures

1.1	Model Selection Between ARIMA and LSTM . . . . .	4
1.2	Advantages of ARIMA and LSTM for hurricane wind speed prediction. . . . .	5
1.3	Structure of the Report . . . . .	7
4.1	Dummy Output Of Code . . . . .	50
5.1	Vertical Workflow: Hurricane Intensity Prediction using ARIMA . . . . .	53
6.1	Vertical KDD Process as Applied in the Project . . . . .	57
6.2	Machine Learning Pipeline for Time-Series Forecasting .	59
7.1	Important Columns for Hurricane Intensity Prediction . .	64
9.1	Forecast vs. Actual Wind Speeds . . . . .	81
9.2	Color-coded visualization of the hurricane wind speed forecasting process . . . . .	82
11.1	Validation checks during configuration parsing . . . . .	103
12.1	Interaction between developer configuration and application interface . . . . .	109
14.1	Visualization of Test Output Results . . . . .	117
15.1	Forecasting Workflow in Deployment Mode . . . . .	124
15.2	Deployment View: Machine Learning Prediction Pipeline	126
15.3	Parameter management flow between the GUI, configuration file, and prediction app . . . . .	128
15.4	Message Handling Flow in Deployment Interface . . . . .	131
16.1	Visualization of Test Output Results . . . . .	135
16.2	Automation Guide Flowchart . . . . .	140
17.1	Model Forecasting Pipeline . . . . .	142
18.1	Lenovo LOQ Essential Gen 9 . . . . .	144
18.2	SanDisk 1TB SSD . . . . .	145

# List of Tables

4.1	Input and Output Specifications . . . . .	48
7.1	Summary of Atlantic Storm Dataset Variables . . . . .	62
11.1	Project Modules and Their Responsibilities . . . . .	101
12.1	Explanation of configuration and model files . . . . .	109
14.1	Explanation of Integration Testing Steps . . . . .	120
15.1	Basic Error Handling Flowchart with Data Checks and Cleaning . . . . .	130
18.1	Hardware Bill of Materials . . . . .	144
18.2	Software and Libraries Used . . . . .	146

# List of Listings

# Acronyms

# 1 Introduction

## 1.1 Problem Description: Hurricane Wind Speed Prediction Using Time Series Models

### 1.1.1 Background

Accurate prediction of hurricane intensity, especially wind speed, is essential for early warning systems, disaster response, insurance risk modeling, and minimizing human and economic losses. As climate volatility increases, so does the importance of tools capable of forecasting wind speed based on prior data patterns.

### 1.1.2 Problem Statement

Traditional models such as the WFL-EMM (Weight Feature Learning-Extensible Markov Model) incorporate genetic algorithms for feature weighting and probabilistic Markov chains to predict storm transitions [Su+11]. While effective in theory, these models are often computationally expensive, hard to tune, and impractical in resource-constrained real-world applications.

There is thus a pressing need for a **simpler, scalable, and interpretable** approach that can accurately predict wind speeds using only historical wind speed data, without excessive preprocessing or domain-specific configuration.

### 1.1.3 Literature Overview

Box et al. [Box+15] laid the groundwork for statistical forecasting with the ARIMA model, which has been widely adopted for time series applications due to its ability to capture trends and autocorrelations. On the other hand, Zhang et al. [ZPH01] reviewed the strengths of neural networks in nonlinear forecasting, highlighting their adaptability but also noting limitations in interpretability and training complexity.

Shi et al. [Shi+15] extended recurrent architectures using Convolutional LSTM for weather forecasting, illustrating that LSTM models

outperform traditional methods in capturing spatio-temporal dependencies. Kordmahalleh et al. [Kor+16] applied sparse recurrent neural networks to hurricane trajectory prediction, demonstrating success with minimal feature engineering. Additionally, Lakshmanan et al. [Lak+07] developed WDSS-II, a system integrating machine learning with weather radar for operational decision support.

Despite these advancements, there remains a gap in producing models that balance accuracy, interpretability, and computational feasibility for wind speed forecasting.

### 1.1.4 Challenges

Based on literature, key challenges in hurricane wind prediction include:

- **Data complexity:** Wind speed patterns are highly nonlinear and affected by chaotic atmospheric dynamics [ZPH01].
- **Model scalability:** Advanced hybrid models often require significant computational power and hyperparameter tuning [Su+11].
- **Deployment feasibility:** Complex architectures such as CNN-LSTM hybrids can be difficult to deploy on embedded or cloud-edge devices [Shi+15].
- **Interpretability:** Many machine learning models (e.g., LSTM) are black boxes, limiting their adoption in mission-critical systems [Box+15].

### 1.1.5 Proposed Solution

To address these issues, this project evaluates and implements one of two distinct time series forecasting approaches:

- **ARIMA (AutoRegressive Integrated Moving Average)** — used for linear trend modeling and interpretable forecasting [Box+15].
- **LSTM (Long Short-Term Memory)** — applied for capturing nonlinear dependencies and long-term memory in time series data [ZPH01].

While both models are explored in the report from a theoretical and practical standpoint, the final implementation is based on only one of these approaches — selected based on the project’s technical constraints, interpretability goals, and dataset characteristics.

This flexibility allows the system to be extended in the future with hybrid or comparative modes, but currently, the deployment pipeline uses either ARIMA or LSTM, not both simultaneously.

### 1.1.6 Model Selection Guide

Choosing between ARIMA and LSTM can feel confusing, but it's just like picking the right tool for the job. Here's a simple way to think about it:

- If your data shows a clear trend or seasonality and you want to understand how the prediction is made, use **ARIMA**. Think of it like using a calculator: fast, explainable, and efficient.
- If your data is messy, complicated, and has patterns that aren't easy to spot, use **LSTM**. It's like using a smart app that learns on its own – powerful but harder to interpret.

**In short:**

- Want simplicity and control? → **ARIMA**
- Want power and flexibility? → **LSTM**

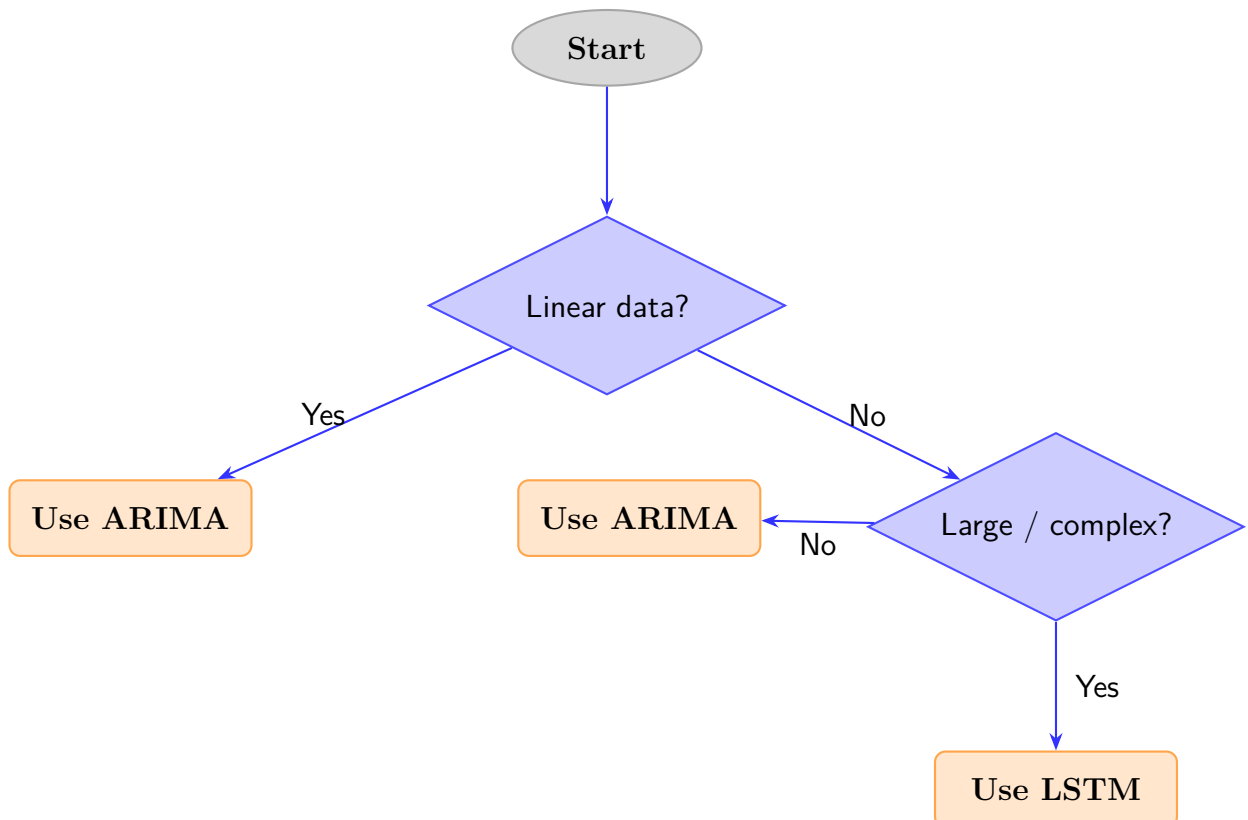


Figure 1.1: Model Selection Between ARIMA and LSTM



### 1.1.7 Advantages

- **Interpretability:** ARIMA provides understandable model parameters.
- **Adaptability:** Both models work across various datasets with minimal preprocessing.
- **Nonlinear capability:** LSTM captures complex temporal dependencies often missed by linear models.
- **Simplicity and scalability:** The combination allows for rapid prototyping and practical deployment.

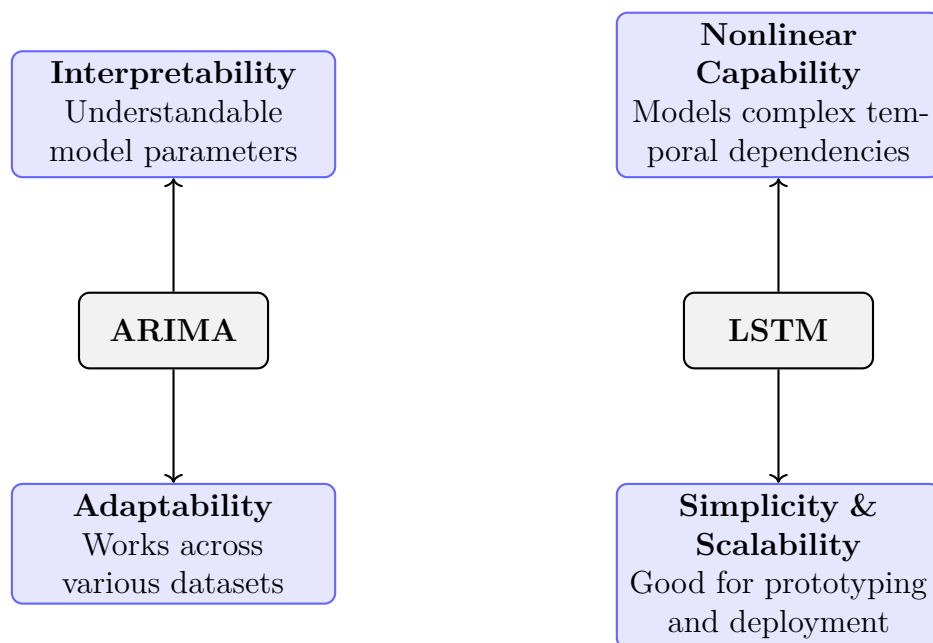


Figure 1.2: Advantages of ARIMA and LSTM for hurricane wind speed prediction.

### 1.1.8 Report Structure

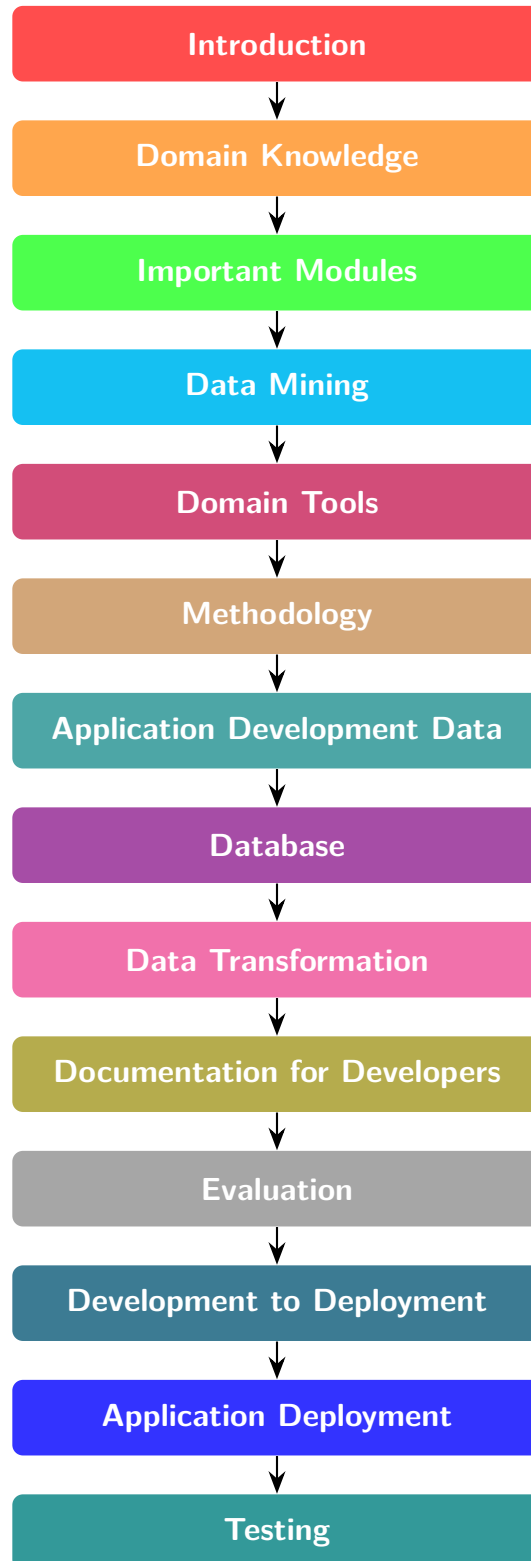
This report is organized into a series of chapters and sections that comprehensively cover the various stages of the project. It starts with the foundational topics, including the introduction, domain knowledge, and essential modules. This is followed by discussions on data mining techniques and domain-specific tools relevant to the project.

The core methodology and data-related chapters detail the technical approach and data processing workflows. Later sections focus on evaluation and monitoring to measure and ensure the system's performance.

The report then describes the deployment workflow, covering the transition from development to production, alongside the testing procedures that guarantee system reliability and robustness.

Finally, an **Application Appendix** is provided, containing key supporting materials such as the Bill of Materials, Software Bill of Materials, the `requirements.txt` file, and a detailed list of packages and tools used throughout the project. This appendix supports reproducibility and ongoing maintenance efforts.

Figure 1.3: Structure of the Report



## 2 Domain Knowledge

### 2.1 Application

Hurricane intensity forecasting plays a critical role in natural disaster preparedness, insurance risk estimation, and infrastructure resilience planning. Our project focuses on using time series forecasting, specifically ARIMA models, to predict the maximum wind speed of Atlantic hurricanes. Accurate predictions can help governments, NGOs, and local communities to prepare resources and evacuations in time, thereby saving lives and reducing economic losses.

### 2.2 Problem

Our group aimed to address the challenging task of hurricane intensity prediction using the NOAA Atlantic Hurricane Database (HURDAT2), sourced from Kaggle. This task presents multiple real-world challenges:

#### 2.2.1 Complex Natural Factors

Hurricane strength depends on numerous interacting environmental elements such as wind shear, sea surface temperature, and atmospheric interactions. Even minor fluctuations in these factors can significantly alter storm intensity [Ema03; KD03].

#### 2.2.2 Imperfect Data

Data collection over oceans is inherently difficult. Inaccuracies in initial values such as wind speed or temperature can lead to substantial forecasting errors, especially for predictions beyond 24–48 hours.

#### 2.2.3 Track vs. Intensity Prediction

While hurricane track forecasting has improved significantly in recent decades, intensity prediction still lags behind. Models show an average error of around 15 mph per day, which poses risks for emergency planning and disaster mitigation [DK+19].

### 2.2.4 Rapid Intensification Events

Some storms, such as Hurricane Maria and Hurricane Dorian, have undergone rapid intensification, where wind speeds increased dramatically in a short time. These events are extremely difficult to predict and can catch communities unprepared [KD03].

### 2.2.5 ARIMA Limitations

Although ARIMA models are proficient at identifying trends and seasonality in time series data, they do not account for the physical processes driving hurricane dynamics. This makes it challenging to predict sudden changes in intensity, especially for rapidly intensifying storms.

## 2.3 Data Acquisition

The dataset used for this study is the `storms.csv` file, sourced from the National Oceanic and Atmospheric Administration (NOAA) via the Atlantic Hurricane Database (HURDAT2), maintained by the National Hurricane Center (NHC) [ON21; Kag21]. It contains meteorological records of Atlantic tropical and subtropical cyclones from 1975 to 2021, collected at 6-hour intervals. The dataset includes 13 attributes:

- Date and time of each storm observation (year, month, day, hour).
- Storm name, such as Amy or Blanche.
- Latitude and longitude of the storm's center, in degrees.
- Storm status, such as tropical depression, tropical storm, or hurricane.
- Saffir-Simpson hurricane category, ranging from 1 to 5 or NA for non-hurricanes.
- Maximum sustained wind speed, in knots.
- Minimum central pressure, in millibars.
- Diameter of tropical storm-force winds, in nautical miles, or NA if not applicable.
- Diameter of hurricane-force winds, in nautical miles, or NA if not applicable.

Preprocessing was conducted to verify the dataset's structure, format, and completeness, ensuring suitability for model building. Given NOAA's reputation and the dataset's use in meteorological research, it is considered a reliable source for forecasting Atlantic storm wind speeds.

## 2.4 Data Quantity

The `storms.csv` dataset provides sufficient scale and granularity for time-series modeling, such as ARIMA. Below is our dataset summary:

### Dataset Summary

Property	Value
Total Records	19,066
Time Period	1975–2021
Frequency	Observations every 6 hours
File Size	1.2 MB
Attributes	13 columns per record

Each storm includes multiple observations, typically every 6 hours, enabling high-resolution time-series modeling and trend analysis for wind speed forecasting.

## 2.5 Data Quality

The dataset, curated by NOAA, is a trusted resource in meteorological research. However, several quality issues were identified and addressed during preprocessing:

### 2.5.1 Missing Values

- Tropical storm-force diameter and hurricane-force diameter: Frequent NA or 0 values, especially in 1975–1976, due to limited historical measurement capabilities.
- Category: NA values for non-hurricane storms, as expected.
- Wind speed, pressure, name, latitude, longitude: No missing values in these core fields.

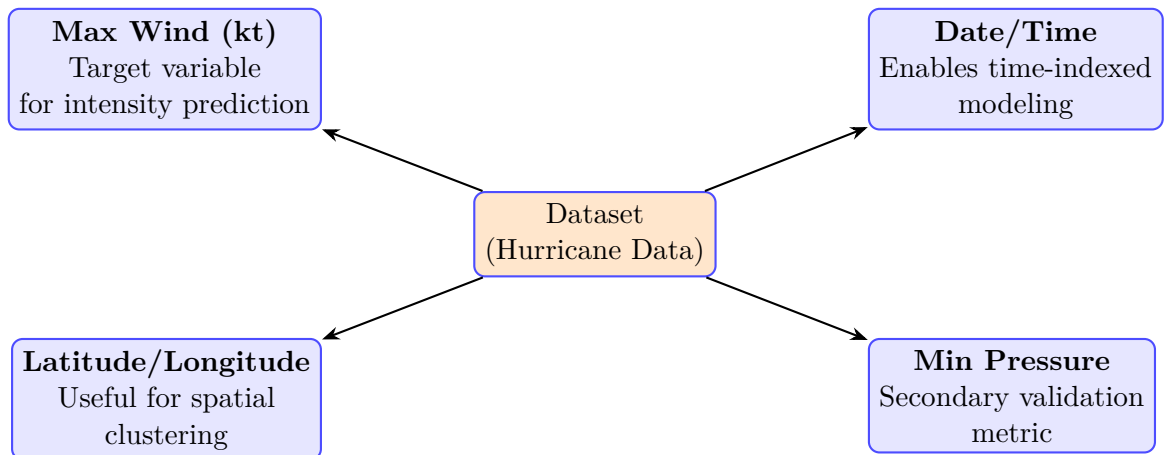
### 2.5.2 Formatting Issues

- Latitude and longitude were provided as numeric values, with positive latitudes for the Northern Hemisphere and negative longitudes for the Western Hemisphere, requiring no conversion.

- Date and time were stored as separate year, month, day, and hour fields, which were combined into a unified timestamp for time-series analysis.

## 2.6 Data Relevance

The dataset is highly relevant to the goal of forecasting wind speeds for Atlantic storms.



- **Wind Speed**: The primary target variable for intensity prediction, ranging from 15 kn to 135 kn.
- **Date and Time**: Enables time-indexed modeling, critical for time-series forecasting.
- **Latitude and Longitude**: Supports potential spatial analysis or tracking storm paths.
- **Pressure**: Serves as a secondary metric to validate wind speed predictions, with lower pressures indicating stronger storms.
- **Status and Category**: Provides context for storm intensity, useful for model interpretation.

## 2.7 Outliers

Outliers were identified in key variables but were generally meteorologically plausible:

- Wind speeds ranged from 15 kn (e.g., Nicholas, 2021-09-17) to 135 kn (e.g., Sam, 2021-09-26), consistent with weak systems and Category 4 hurricanes, respectively.

- Pressures ranged from 927 mbar (e.g., Sam, 2021-09-26) to 1015 mbar (e.g., Dottie, 1976-08-21), typical for strong hurricanes and weak depressions.
- Extreme latitudes, such as  $58.1^{\circ}\text{N}$  (e.g., Gladys, 1975-10-03), and longitudes, such as  $-20.6^{\circ}\text{W}$  (e.g., Larry, 2021-08-31), reflect northern or extratropical storm paths.

Extreme values were reviewed to confirm they represented valid meteorological events, such as rapid intensification, rather than data entry errors.

## 2.8 Anomalies

Several anomalies were detected in the dataset:

- The dataset is truncated at Wanda on 2021-11-08, with an incomplete final entry (“36.8...”), suggesting a parsing or truncation error.
- Rapid changes, such as Sam’s wind speed increasing from 60 kn to 135 kn in 36 h (2021-09-24 to 2021-09-26), are plausible for major hurricanes.
- Abrupt status transitions, such as Odette shifting from “other low” to “tropical storm” on 2021-09-17, may indicate classification changes.
- Constant pressure values, such as Amy at 1013 mbar across multiple entries in 1975, suggest coarse historical measurements.

These anomalies were addressed by cleaning the dataset, removing incomplete entries, and filling minor gaps in wind speed with preceding or following values, ensuring a reliable input for the ARIMA model.

## 2.9 Summary

This chapter highlights the characteristics and challenges of the NOAA HURDAT2 dataset for hurricane wind speed forecasting. Despite its high quality and relevance, the dataset contains imperfections, such as missing values in storm size measurements, truncation errors, and historical data limitations. Addressing these issues through preprocessing and augmentation is essential for building effective time-series models. These considerations emphasize the complexity of hurricane forecasting and its importance for disaster preparedness.



## 3 Important Modules

All required Python modules for this project can be freely downloaded and installed from the official Python Package Index (PyPI) [Pyt25].

### 3.1 Pandas

#### 3.1.1 Pandas Overview

The **pandas** library is a cornerstone of Python-based data science, delivering powerful, intuitive, and high-performance tools for analyzing and manipulating structured datasets [McK11]. Tailored for efficiency, it provides flexible data structures that streamline complex data operations, making it indispensable for tasks like preprocessing time-series wind speed data in `model_utils.py` and formatting forecast outputs in `app.py`. Its expressive design empowers users to handle meteorological data with ease, aligning with the project's goal of generating accurate hurricane intensity predictions. By offering a robust framework for data wrangling, **pandas** ensures seamless integration into the data pipeline, from ingestion to analysis.

#### 3.1.2 Pandas Functionality

Central to **pandas** are its **Series** and **DataFrame** structures, which facilitate efficient manipulation of labeled, tabular data, offering functionality comparable to R and SQL environments [McK11]. The **DataFrame**, a two-dimensional table, supports operations such as sorting, filtering, and missing value imputation, as utilized in `model_utils.py` to clean wind speed datasets. The **Series**, a one-dimensional array with labeled indices, enables precise time-series handling, critical for ARIMA and LSTM inputs in `train_models.ipynb`. These structures allow for intuitive data operations, such as chronological sorting of timestamps and coercion of non-numeric values, ensuring data integrity. In the project, **pandas** underpins the preprocessing pipeline, enabling robust data preparation and output generation in `app.py`, enhancing the reliability of hurricane forecasts.

### 3.1.3 Installing Pandas

This guide helps you install the pandas library, a fantastic tool for working with data in Python. If you're brand new to coding, don't worry—we'll go step by step to get pandas up and running on your computer. You'll need Python installed, and we'll use a tool called pip to download pandas. Let's jump in and make it happen!

First, check if Python is installed on your computer. Open a terminal—on Windows, this could be Command Prompt or PowerShell; on Mac or Linux, use the Terminal app. Type `python -version` and press Enter. If you see a version number, like Python 3.8 or higher, you're all set. If you get an error, head to <https://www.python.org/downloads/>, download the latest Python version, and follow the installation steps. During setup, make sure to check the option to add Python to your system PATH so you can run it from the terminal easily.

Next, confirm that pip, Python's package installer, is ready to go. In the terminal, type `pip -version` and press Enter. If a version number pops up, pip is good to use. If not, you'll need to install pip. Visit <https://bootstrap.pypa.io/get-pip.py>, download the `get-pip.py` file, save it to your computer, and then run `python get-pip.py` in the terminal from the folder where you saved the file. This sets up pip, letting you install libraries like pandas.

Now it's time to install pandas! In your terminal, type this command and press Enter:

```
pip install pandas
```

This tells pip to fetch pandas and any extra bits it needs, like another library called numpy. You'll see some text scrolling by as it downloads and installs. When it's done, pandas is ready to use on your computer.

To make sure pandas installed correctly, open a Python session by typing `python` in the terminal and pressing Enter. Then type `import pandas as pd` and press Enter. If nothing goes wrong (no error messages), pandas is working! Try typing `pd.Series([1, 2, 3])` and press Enter to see a small list of numbers displayed as a pandas Series, a cool way to store data. When you're done, type `exit()` and press Enter to close Python.

If something doesn't work, check your internet connection, as pip needs it to download pandas. You can also try updating pip by typing `pip install -upgrade pip` and pressing Enter, then retry the installation. If you're still stuck, make sure you're using a recent Python version, like 3.8 or later, since pandas works best with those. Once pandas is installed, you're ready to explore data in Python.

### 3.1.4 Example: Using Pandas

This subsection demonstrates how to utilize the **pandas** library to load data from a comma-separated values (CSV) file into a **DataFrame** and perform essential operations such as filtering and aggregation. These operations are critical for analyzing structured datasets, enabling users to extract valuable insights. The examples are designed for beginners, offering step-by-step Python code with explanations. For clarity, the code uses **readcsv** as a simplified representation of the **pandas read\_csv** method and avoids underscores in variable names. The examples assume **pandas** is installed and use a sample CSV file with columns for product names, prices, and quantities.

Consider a CSV file named **datafile.csv** with the following content:

```
product,price,quantity
Laptop,999.99,5
Phone,499.99,10
Tablet,299.99,8
```

The first example illustrates loading this file into a **DataFrame** using **pandas**.

Listing 3.1: Loading a CSV file into a DataFrame

```
import pandas as pd

# Load the CSV file into a DataFrame
df = pd.readcsv('datafile.csv')

# Display the DataFrame
print(df)
```

This code imports **pandas** with the alias **pd**, loads **datafile.csv** into a **DataFrame** named **df**, and prints its contents. The output displays a table with columns **product**, **price**, and **quantity**, mirroring the CSV structure. The **readcsv** method (representing **read\_csv**) interprets the first row as column headers, simplifying data access.

The second example shows filtering the **DataFrame** to select rows where the price exceeds 400. Filtering allows users to focus on specific data subsets based on conditions.

Listing 3.2: Filtering rows in a DataFrame

```
import pandas as pd

# Load the CSV file
df = pd.readcsv('datafile.csv')

# Filter rows where price is greater than 400
expensiveproducts = df[df['price'] > 400]

# Display the filtered DataFrame
print(expensiveproducts)
```

The condition `df['price'] > 400` creates a boolean mask, and `df[...]` selects matching rows. The resulting **DataFrame**, **expensiveproducts**, includes only rows for **Laptop** and **Phone**, with prices of 999.99 and 499.99, respectively. This technique is useful for isolating relevant data points, such as high-value items.

The third example performs aggregation, calculating the average price and total quantity across all products. Aggregation summarizes data, providing insights into overall trends.

Listing 3.3: Aggregating data in a DataFrame

```
import pandas as pd

# Load the CSV file
df = pd.readcsv('datafile.csv')

# Compute the average price and total quantity
averageprice = df['price'].mean()
totalquantity = df['quantity'].sum()

# Display the results
print(f"Average Price: ${averageprice:.2f}")
print(f"Total Quantity: {totalquantity}")
```

The **mean** method calculates the average of the **price** column (599.99), and the **sum** method totals the **quantity** column (23). The results are formatted for clarity using string formatting. These operations enable users to derive summary statistics, essential for understanding dataset characteristics.

These examples provide beginners with foundational skills to load, filter, and aggregate data using **pandas**. Users can extend these techniques to handle larger datasets or perform more complex analyses, advancing their data science proficiency.

### 3.1.5 Example – Manual

```
import pandas as pd
df = pd.DataFrame({'a':[1,2,3], 'b':[4,5,6]})
df = df[df['a'] > 1]
```

### 3.1.6 Example – Code

Listing 3.4: Grouping and aggregating data

```
import pandas as pd

# Create a DataFrame from a dictionary
data = pd.DataFrame({'category': ['A', 'B', 'A', 'B'], 'value':
    ↪ [10, 20, 30, 40]})

# Group by category and sum the value column
```

```
groupsum = data.groupby('category').agg({'value': 'sum'})

# Display the grouped DataFrame
print(groupsum)
```

### 3.1.7 Example – Files

Loading and saving data in various formats:

```
# CSV
df.to_csv('out.csv', index=False)
df = pd.read_csv('out.csv')

# Excel
df.to_excel('out.xlsx', sheet_name='Sheet1', index=False)
df = pd.read_excel('in.xlsx', sheet_name='Sheet1')

# JSON
df.to_json('data.json', orient='records', lines=True)
df = pd.read_json('data.json', lines=True)

# Pickle (for saving/loading Python objects)
df.to_pickle('data.pkl')
df = pd.read_pickle('data.pkl')

# SQL (using sqlite3 as an example)
import sqlite3
conn = sqlite3.connect('example.db')
df.to_sql('table_name', conn, if_exists='replace', index=False)
df = pd.read_sql('SELECT * FROM table_name', conn)
```

### 3.1.8 Further Reading

Wes McKinney, *pandas: a Foundational Python Library for Data Analysis and Statistics* [McK11].

## 3.2 Statsmodels

### 3.2.1 Introduction

Statsmodels is an open-source Python library designed for econometric and statistical modeling, hypothesis testing, and data exploration [SP10]. It complements the scientific Python ecosystem by providing classes and functions to estimate many different statistical models, perform statistical

tests, and conduct data exploration and visualization. Statsmodels is widely used in academic research and industry for its robustness, comprehensive statistical output, and R-like interface, making it particularly valuable for users with backgrounds in statistics and econometrics.

### 3.2.2 Description

Statsmodels provides extensive capabilities in the following areas:

- **Regression Models:** Supports a broad spectrum of regression techniques including Ordinary Least Squares (OLS), Generalized Least Squares (GLS), Weighted Least Squares (WLS), robust regression methods, and Generalized Linear Models (GLM) such as logistic and Poisson regression.
- **Time-Series Analysis:** Implements classic time-series models such as Autoregressive Integrated Moving Average (ARIMA), Seasonal ARIMA (SARIMA), Vector Autoregression (VAR), state space models, and structural time series.
- **Statistical Tests and Diagnostics:** Includes tools for hypothesis testing (e.g., t-tests, F-tests), diagnostic tests for autocorrelation (Durbin-Watson), heteroscedasticity (Breusch-Pagan), normality (Jarque-Bera), and multicollinearity, among others.
- **Model Evaluation and Inference:** Detailed summaries of model fits are provided including coefficient estimates, standard errors, confidence intervals, p-values, goodness-of-fit statistics (R-squared, AIC, BIC), and residual analysis.
- **Formula Interface:** Similar to R's formula syntax, allowing users to specify models in a concise and readable way using strings.
- **Integration:** Seamlessly integrates with pandas DataFrames for data handling, and with NumPy and SciPy for numerical and scientific computation.

### 3.2.3 Installation

Installing Statsmodels is straightforward but, as a new user, here are some step-by-step tips to make sure everything goes smoothly:

1. **Install Python:** Make sure you have Python installed on your computer. You can download it from <https://www.python.org/downloads/>. For most users, Python 3.8 or higher is recommended.

2. **Set up a Virtual Environment (Recommended):** It's good practice to create an isolated environment for your projects to avoid conflicts between different Python packages.

```
python -m venv myenv
```

This creates a new environment named **myenv**. Activate it with:

- On Windows:  

```
myenv\Scripts\activate
```
- On macOS/Linux:  

```
source myenv/bin/activate
```

3. **Upgrade pip:** Before installing, upgrade pip (Python's package manager) to the latest version:

```
pip install --upgrade pip
```

4. **Install Statsmodels:** Now you can install Statsmodels using pip:

```
pip install statsmodels
```

This command will also install the required dependencies like **numpy**, **scipy**, and **pandas** if you don't have them already.

5. **Verify Installation:** After installation, check if Statsmodels is correctly installed by running Python and importing it:

```
python
>>> import statsmodels.api as sm
>>> print(sm.__version__)
```

If no errors appear and the version number prints, you are ready to go!

6. **Optional - Using Anaconda:** If you are using the Anaconda Python distribution, Statsmodels can be installed via conda:

```
conda install statsmodels
```

This is often easier for beginners as it handles all dependencies automatically.

By following these steps, even if you are new to Python and package installation, you will be able to set up Statsmodels and start your statistical modeling smoothly.

### 3.2.4 Example – Description

Statsmodels shines in providing comprehensive statistical analysis with rich diagnostic information, unlike many machine learning libraries focused primarily on prediction accuracy. Its typical workflow includes:

1. Specifying a model either through matrices (endogenous and exogenous variables) or via formula strings.
2. Fitting the model to data, estimating parameters using maximum likelihood or least squares.
3. Examining detailed output reports to interpret coefficients, test hypotheses, and diagnose model adequacy.
4. Using fitted models for prediction, forecasting, and simulation.

Common modeling scenarios include:

- **Linear Regression:** Understand the relationship between a dependent variable and one or more independent variables.
- **Time-Series Forecasting:** Model and forecast data collected over time, accounting for trends, seasonality, and autocorrelation.
- **Categorical Outcome Modeling:** Logistic regression for binary or multinomial outcomes.

### 3.2.5 Example – Manual

Here are some fundamental examples illustrating Statsmodels usage on a dataset `df` containing variables `a`, `b`, `c`, and a binary `outcome`:

```
# Example 1: Linear regression using OLS (matrix interface)
import statsmodels.api as sm
X = sm.add_constant(df[['a']]) # Adds intercept term
model = sm.OLS(df['b'], X).fit()
print(model.summary())
```

```
# Example 2: Time-Series ARIMA model
from statsmodels.tsa.arima.model import ARIMA
model = ARIMA(df['b'], order=(1,1,1))
fit = model.fit()
print(fit.summary())
```

```
# Example 3: Logistic regression with formula API
import statsmodels.formula.api as smf
logit_model = smf.logit('outcome ~ a + c', data=df).fit()
print(logit_model.summary())
```



Each of these examples demonstrates how Statsmodels provides detailed statistical summaries including parameter estimates, confidence intervals, and diagnostic statistics.

### 3.2.6 Example – Code

More advanced examples combining model fitting, prediction, and diagnostic testing:

```
# Example 1: Linear regression with formula and prediction
import statsmodels.formula.api as smf
fit = smf.ols('b ~ a + c', data=df).fit()
print(fit.summary())
predictions = fit.predict(df[['a', 'c']])

# Example 2: Seasonal ARIMA for forecasting
from statsmodels.tsa.statespace.sarimax import SARIMAX
model = SARIMAX(df['b'], order=(1,1,1), seasonal_order=(1,1,1,12))
results = model.fit()
forecast = results.get_forecast(steps=5)
print(forecast.summary_frame())

# Example 3: Conducting heteroscedasticity test on residuals
from statsmodels.stats.diagnostic import het_breuschpagan
lm_test = het_breuschpagan(fit.resid, fit.model.exog)
print('Breusch-Pagan test statistic:', lm_test[0])
print('p-value:', lm_test[1])
```

These examples highlight the versatility of Statsmodels, ranging from predictive modeling to rigorous statistical diagnostics.

### 3.2.7 Example – Files

Statsmodels natively supports working with pandas DataFrames, facilitating smooth data handling. For persistent storage, models can be saved and reloaded using Python's serialization libraries such as **pickle** or **joblib**, enabling workflows that require saving fitted models for future predictions or sharing.

Example of saving and loading a fitted model:

```
import pickle

# Save model to disk
with open('linear_model.pkl', 'wb') as f:
    pickle.dump(fit, f)
```

```
# Load model from disk
with open('linear_model.pkl', 'rb') as f:
    loaded_model = pickle.load(f)

# Use loaded model to predict new data
new_predictions = loaded_model.predict(new_df[['a',
        'c']])
```

Additionally, Statsmodels supports exporting results to tables compatible with LaTeX or HTML for easy reporting.

### 3.2.8 Further Reading

Seabold, S., & Perktold, J. (2010). *Statsmodels: Econometric and Statistical Modeling with Python*. Proceedings of the 9th Python in Science Conference [SP10] provides a detailed overview and motivation behind Statsmodels' development and capabilities.

## 3.3 TensorFlow

### 3.3.1 Introduction

TensorFlow is an open-source framework developed by Google for large-scale machine learning and deep learning applications. It allows users to design, train, and deploy machine learning models using flexible and efficient dataflow graphs [Aba+16]. TensorFlow supports both research experimentation and production deployment, enabling models to run on a variety of hardware, from personal devices to distributed clusters.

### 3.3.2 Description

TensorFlow's core strength lies in its ability to represent complex computations as dataflow graphs where nodes correspond to mathematical operations, and edges represent multidimensional data arrays called tensors. It provides extensive support for:

- **Distributed Computing:** TensorFlow can automatically distribute computation across CPUs, GPUs, and specialized hardware like TPUs (Tensor Processing Units), optimizing performance for large datasets and complex models.
- **Deep Neural Networks:** Includes high-level APIs such as `tf.keras` to build and train deep learning models with layers like convolutional, recurrent, and dense layers.

- **Eager Execution:** TensorFlow 2.x enables dynamic computation, allowing immediate evaluation of operations, which simplifies debugging and accelerates model development.
- **Data Pipelines:** Tools like `tf.data` provide efficient input pipelines to preprocess and feed data for training.
- **Model Deployment:** Facilities to save, export, and deploy models across platforms including mobile, web, and cloud services.

### 3.3.3 Installation

To install TensorFlow, follow these step-by-step instructions designed especially for users new to Python and machine learning:

1. **Prerequisites:** Make sure you have Python 3.7 or later installed on your computer. You can download it from <https://www.python.org/downloads/>. To confirm your Python version, open your command prompt (Windows) or terminal (Linux/macOS) and run:

```
python --version
```

2. **Set Up a Virtual Environment (Recommended):** It is a good practice to create a virtual environment to keep your Python packages organized and prevent conflicts:

```
python -m venv tensorflow-env
```

Activate it by running:

- Windows:

```
tensorflow-env\Scripts\activate
```

- macOS/Linux:

```
source tensorflow-env/bin/activate
```

3. **Install TensorFlow via pip:** With your virtual environment activated, run:

```
pip install tensorflow
```

This will install the latest stable version of TensorFlow along with necessary dependencies.

4. **Verify the installation:** After installation, check if TensorFlow is installed correctly by launching Python and importing the library:

```
python
>>> import tensorflow as tf
>>> print(tf.__version__)
```

If you see the version number printed without errors, you are good to go!

5. **GPU Support (Optional):** If your system has compatible NVIDIA GPUs and you want faster training, install the GPU-enabled TensorFlow package:

```
pip install tensorflow-gpu
```

Note that this requires installing NVIDIA CUDA and cuDNN drivers, which you can find instructions for at <https://www.tensorflow.org/install/gpu>.

6. **Using Anaconda (Alternative):** If you use the Anaconda Python distribution, you can install TensorFlow easily with:

```
conda install tensorflow
```

This automatically manages dependencies for you.

### 3.3.4 Example – Description

TensorFlow uses two main execution paradigms:

- **Graph Execution (TensorFlow 1.x):** Users define a static computational graph, then run the graph in a session. This approach optimizes computations but is less intuitive for debugging.
- **Eager Execution (TensorFlow 2.x):** Operations are evaluated immediately as they are called, enabling easier and more interactive model building, similar to standard Python programming.

TensorFlow models typically consist of these steps:

1. Define the computational graph or model architecture (e.g., layers in a neural network).
2. Compile the model by specifying optimizer, loss function, and evaluation metrics.
3. Train the model on data using `model.fit()`.
4. Evaluate or predict outcomes on new data using `model.evaluate()` or `model.predict()`.

Examples include:

- **Matrix multiplication using tensors:** Performing operations on multidimensional arrays to compute mathematical results efficiently.
- **Building a simple feedforward neural network for regression:** Predicting continuous outputs such as house prices based on input features.
- **Implementing a convolutional neural network (CNN) for image classification:** Identifying objects in images, such as classifying handwritten digits or detecting cats and dogs.

### 3.3.5 Example – Manual

Basic code examples illustrating key TensorFlow operations:

```
# Example 1: Simple Tensor Operation (Matrix Multiplication)
import tensorflow as tf
x = tf.constant([[1.0, 2.0]])
y = tf.matmul(x, x, transpose_b=True)
print(y.numpy())
# Output: [[5.]]
```

```
# Example 2: Define a simple sequential model for regression
model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation='relu', input_shape=(5,)),
    tf.keras.layers.Dense(1)
])
model.compile(optimizer='adam', loss='mse')

# Example 3: Train the model (assuming X_train, y_train are defined)
model.fit(X_train, y_train, epochs=5, batch_size=32)
```

### 3.3.6 Example – Code

More detailed examples including evaluation and prediction:

```
# Example 1: Predicting new values using the trained model
y_pred = model.predict(X_test)
print(y_pred)

# Example 2: Evaluate the model performance on test data
loss = model.evaluate(X_test, y_test)
print(f'Test_loss:_{loss}')

# Example 3: Using tf.data to create a dataset pipeline for efficient training
import numpy as np
dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
dataset = dataset.shuffle(buffer_size=1024).batch(32)

# Train the model using the dataset pipeline
model.fit(dataset, epochs=10)
```

### 3.3.7 Example – Files

TensorFlow allows saving and loading models and datasets for reuse:

```
# Save the entire model to a HDF5 file
model.save('model.h5')

# Load the model later from HDF5 file
new_model = tf.keras.models.load_model('model.h5')

# Save model in TensorFlow SavedModel format (recommended for deployment)
model.save('saved_model_dir')
```

```
# Load SavedModel format
loaded_model = tf.keras.models.load_model('saved_model_dir')

# Working with tf.data datasets:
# Save dataset as TFRecord files for efficient storage and input
def _bytes_feature(value):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

# (User-defined code required for TFRecord creation and reading)
```

Additional use cases include exporting models to TensorFlow Lite for mobile deployment or to TensorFlow.js for web applications.

### 3.3.8 Further Reading

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Zheng, X., *TensorFlow: A System for Large-Scale Machine Learning*, Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016 [Aba+16].

## 3.4 NumPy

### 3.4.1 Introduction

NumPy, short for Numerical Python, is a powerful open-source library for numerical computing in Python. It is optimized for performance and serves as the foundation for many scientific and data analysis packages [Har+20]

### 3.4.2 Description

As described by Harris et al. [Har+20], NumPy introduces the **ndarray**, a multidimensional array object. Unlike regular Python lists, NumPy arrays support element-wise operations and broadcasting, which enable fast computations over large datasets. NumPy has become widely used across fields such as data science, machine learning, physics, and engineering.

### 3.4.3 Installation

If you're new to Python or programming, follow these step-by-step instructions to install NumPy correctly on your system.

### 1. Check if Python is installed:

Open your command prompt (Windows) or terminal (Mac/Linux), and type:

```
python --version
```

If Python is installed, you'll see something like **Python 3.10.12**. If not, download and install Python from <https://www.python.org/downloads/>, and make sure to check the box **Add Python to PATH** during installation.

### 2. Ensure pip is installed:

Pip is Python's package installer. You can check and install it using:

```
python -m ensurepip --upgrade
```

This ensures pip is available and up to date.

### 3. Install NumPy using pip:

Now you're ready to install NumPy:

```
pip install numpy
```

This command will download and install the latest version of NumPy from the Python Package Index (PyPI).

### 4. Verify the installation:

Launch Python in your terminal or shell and run:

```
python
>>> import numpy as np
>>> print(np.__version__)
```

This should display the installed NumPy version, such as **1.26.4**, confirming that NumPy is working correctly.



**Tip:** If you're using Anaconda or Jupyter Notebook, you can also install NumPy using:

```
conda install numpy
```

### 3.4.4 Example - Description

These examples show what NumPy is commonly used for:

- **Statistical Analysis:** Calculate mean, median, standard deviation.
- **Matrix Operations:** Dot product, matrix multiplication, transpose.
- **Array Manipulations:** Reshaping, stacking, flattening arrays.

### 3.4.5 Example - Manual

Step-by-step breakdowns:

#### 1. Mean and Standard Deviation

```
data = np.array([4, 7, 1, 9])
mean = np.mean(data)
std = np.std(data)
```

#### 2. 2D Array Matrix Multiplication

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = np.dot(A, B)
```

#### 3. Broadcasting Example

```
arr = np.array([1, 2, 3])
print(arr + 5)
```

### 3.4.6 Example - Code

Listing 3.5: 1. Normalize Data

```
import numpy as np
data = np.array([10, 20, 30, 40, 50])
normalized = (data - np.mean(data)) / np.std(data)
print("Normalized:", normalized)
```

Listing 3.6: 2. Matrix Multiplication

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])
result = np.dot(A, B)
print("Dot product:\n", result)
```

Listing 3.7: 3. Broadcasting Addition

```
arr = np.array([1, 2, 3])
added = arr + 10
print("After adding 10:", added)
```

### 3.4.7 Example - Files

Listing 3.8: 1. Save and Load Array

```
import numpy as np
arr = np.array([100, 200, 300])
np.save('saved_array.npy', arr)
loaded = np.load('saved_array.npy')
print("Loaded array:", loaded)
```

Listing 3.9: 2. Save and Load Multiple Arrays

```
np.savez('multiple_arrays.npz', x=arr, y=arr*2)
data = np.load('multiple_arrays.npz')
print("x:", data['x'], "y:", data['y'])
```

Listing 3.10: 3. Save as Text File

```
np.savetxt('data.txt', arr, delimiter=',')
loaded_text = np.loadtxt('data.txt', delimiter=',')
print("Text file loaded:", loaded_text)
```

### 3.4.8 Further Reading

- Official Documentation: <https://numpy.org/doc/>
- Beginner Tutorials: <https://numpy.org/learn/>
- NumPy Quickstart: <https://numpy.org/doc/stable/user/quickstart.html>

## 3.5 Matplotlib

### 3.5.1 Introduction

Matplotlib is the de facto standard library for creating 2D plots and visualizations in Python [Hun07]. It offers a powerful, flexible, and easy-to-use framework for generating a wide variety of static, animated, and interactive plots. Due to its extensive functionality and broad adoption in scientific computing, data analysis, and machine learning communities, Matplotlib is considered essential for visualizing data and conveying insights clearly and effectively.

### 3.5.2 Description

Matplotlib supports numerous plot types, including but not limited to:

- **Line plots:** Useful for showing trends, continuous data, or time series.
- **Scatter plots:** Ideal for illustrating relationships or correlations between variables.
- **Histograms:** Depict data distribution by grouping values into bins.
- **Bar charts:** Used for categorical comparisons.
- **Pie charts:** Visualize proportions within a whole.
- **Subplots:** Enable multiple plots within a single figure for comparative visualization.

Matplotlib also provides fine-grained control over plot elements such as colors, labels, gridlines, legends, fonts, and figure size. Additionally, it supports exporting visualizations to various file formats, including PNG, PDF, SVG, and EPS, making it suitable for publication-quality graphics [Hun07].

### 3.5.3 Installation

For users who are new to Python and Matplotlib, the installation process involves a few simple steps:

1. **Ensure Python is installed:** Download and install Python from the official website <https://www.python.org/downloads/> if it is not already present on your system.
2. **Open a command prompt or terminal:**

- Windows: Press **Win + R**, type **cmd**, and press Enter.
- Linux/macOS: Open the Terminal application.

**3. Verify Python installation by typing:**

```
python --version
```

or, on some systems,

```
python3 --version
```

**4. Install Matplotlib using pip:**

```
pip install matplotlib
```

**5. If you encounter permission issues, try:**

```
pip install --user matplotlib
```

**6. Verify the installation by opening Python and importing Matplotlib:**

```
python  
>>> import matplotlib  
>>> matplotlib.__version__
```

**7. If you use the Anaconda distribution, Matplotlib often comes pre-installed. Update it with:**

```
conda update matplotlib
```

### 3.5.4 Example – Description

Matplotlib offers two primary interfaces for creating plots:

- **Pyplot interface (imperative):** Provides a MATLAB-like, stateful interface suitable for quick and simple plotting commands.
- **Object-oriented interface:** Uses Figure and Axes objects to build plots, allowing for greater control and customization, especially for complex figures or multiple subplots.

Both approaches can be used interchangeably, depending on the complexity of the visualization and user preference.

### 3.5.5 Example – Manual

A straightforward example using the Pyplot interface to create a simple line plot, add a title, save it as an image file, and display it is shown below:

```
import matplotlib.pyplot as plt

# Define data points
x = [1, 2, 3]
y = [4, 5, 6]

# Create a line plot
plt.plot(x, y)

# Add a title to the plot
plt.title('Example Plot')

# Save the figure as a PNG image
plt.savefig('fig.png')

# Display the plot
plt.show()
```

### 3.5.6 Example – Code

Below are several examples demonstrating different plot types using the object-oriented approach for better customization:

```
import matplotlib.pyplot as plt

# Histogram Example
data = [12, 15, 13, 17, 19, 14, 16, 18, 20, 13, 15, 16]
fig, ax = plt.subplots()
ax.hist(data, bins=20, color='skyblue')
ax.set_xlabel('Value')
ax.set_ylabel('Frequency')
ax.set_title('Histogram Example')
plt.show()

# Scatter Plot Example
fig, ax = plt.subplots()
x = [5, 7, 8, 7, 2, 17, 2, 9]
y = [99, 86, 87, 88, 100, 86, 103, 87]
ax.scatter(x, y, color='red')
ax.set_title('Scatter Plot Example')
plt.show()

# Bar Chart Example
fig, ax = plt.subplots()
categories = ['A', 'B', 'C']
values = [10, 20, 15]
ax.bar(categories, values, color='green')
ax.set_title('Bar Chart Example')
plt.show()
```

### 3.5.7 Example – Files

Matplotlib plots can be embedded directly within interactive environments such as Jupyter notebooks for inline visualization, making data exploration intuitive and dynamic. Additionally, plots can be saved to external files using the `plt.savefig()` function. Supported file formats include:

- **PNG** — Portable Network Graphics, suitable for web and general use.
- **PDF** — Portable Document Format, preferred for high-quality print and documentation.

- **SVG** — Scalable Vector Graphics, ideal for scalable, lossless images in web or graphic design.

To save a figure as a PDF file, use:

```
plt.savefig('plot.pdf')
```

This flexibility allows seamless integration of plots into reports, presentations, and publications.

### 3.5.8 Further Reading

Hunter, J. D., *matplotlib – A Portable Python Plotting Package*, Computing in Science & Engineering, 9(3), 90–95 (2007) [Hun07].

## 3.6 Scikit-learn

### 3.6.1 Introduction

Scikit-learn is a widely used, user-friendly Python library that provides a comprehensive suite of machine learning algorithms and tools [PVG+11]. It is designed for efficient and accessible application of machine learning techniques such as classification, regression, clustering, and dimensionality reduction. Scikit-learn emphasizes ease of use and consistent API design, making it an essential tool for both beginners and experienced practitioners in data science and artificial intelligence.

### 3.6.2 Description

Scikit-learn offers a broad array of machine learning functionalities, including:

- **Classification:** Algorithms such as Logistic Regression, Support Vector Machines (SVM), Decision Trees, and Random Forests used for categorizing data into predefined classes.
- **Regression:** Models including Linear Regression, Ridge Regression, and Gradient Boosting for predicting continuous outcomes.
- **Clustering:** Techniques like K-Means, DBSCAN, and Agglomerative Clustering for grouping data without labeled outcomes.

- **Dimensionality Reduction:** Methods such as Principal Component Analysis (PCA) and t-SNE to reduce feature space while preserving important information.
- **Model Selection and Evaluation:** Tools for hyperparameter tuning, cross-validation, and performance metrics to optimize and assess models.

All these algorithms share a consistent, easy-to-learn interface with core methods like `fit()`, `predict()`, and `score()`, facilitating smooth workflows from training to evaluation [PVG+11].

### 3.6.3 Installation

For users new to Python or machine learning, follow these clear step-by-step instructions to install scikit-learn:

1. **Verify Python installation:** Make sure Python 3.6 or later is installed on your computer. You can download it from <https://www.python.org/downloads/> if it's not already installed.
2. **Open your command prompt or terminal:**
  - **Windows:** Press Win + R, type `cmd`, and press Enter.
  - **Linux/macOS:** Open the Terminal application.
3. **Check your Python version:** Run the following command to confirm your Python version:

```
python --version
```

This helps ensure compatibility with scikit-learn.

4. **(Recommended) Create a virtual environment:** To avoid package conflicts, create and activate a virtual environment:

```
python -m venv sklearn-env
```

Activate it by running:

- Windows:



```
sklearn-env\Scripts\activate
```

- macOS/Linux:

```
source sklearn-env/bin/activate
```

5. **Install scikit-learn using pip:** Once your environment is ready, install scikit-learn by running:

```
pip install scikit-learn
```

6. **Handling permission errors:** If you encounter permission errors, try installing with the `-user` flag:

```
pip install --user scikit-learn
```

7. **Verify the installation:** Open Python and import scikit-learn to check that it installed correctly:

```
python
>>> import sklearn
>>> print(sklearn.__version__)
```

8. **Using Anaconda (alternative method):** If you are using the Anaconda distribution, scikit-learn is often pre-installed. To update or install it, use:

```
conda install scikit-learn
```

or

```
conda update scikit-learn
```

You can learn more about Anaconda at <https://www.anaconda.com/products/distribution>.

### 3.6.4 Example – Description

Scikit-learn follows a straightforward approach to model training and prediction:

- `fit(X, y)` is used to train the model on the input features `X` and target values `y`.
- `predict(X)` applies the trained model to new input data `X` to generate predictions.
- `score(X, y)` evaluates the model's performance on test data.

This uniform API allows easy switching between different algorithms with minimal code changes.

Examples include:

- **Classification:** Train a Support Vector Machine to classify handwritten digits.
- **Regression:** Fit a Ridge Regression model to predict housing prices.
- **Clustering:** Apply K-Means clustering to group customers based on purchasing behavior.

### 3.6.5 Example – Manual

Basic examples for model training in different tasks:

```
# Linear Regression (Regression task)
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)

# Logistic Regression (Classification task)
```

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X_train, y_train)

# K-Means Clustering (Unsupervised task)
from sklearn.cluster import KMeans
model = KMeans(n_clusters=3)
model.fit(X_train)
```

### 3.6.6 Example – Code

Generating predictions and evaluating model performance for different scenarios:

```
from sklearn.metrics import mean_squared_error, accuracy_score

# Regression: Mean Squared Error (MSE)
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

```
# Classification: Accuracy Score
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')
```

```
# Clustering: Compute silhouette score to evaluate clustering quality
from sklearn.metrics import silhouette_score
labels = model.labels_
score = silhouette_score(X_test, labels)
print(f'Silhouette Score: {score}')
```

### 3.6.7 Example – Files

Saving and loading models to disk for reuse and deployment:

```
import joblib
```

```
# Save the trained model to a file
joblib.dump(model, 'model.pkl')

# Load the model from the file
model = joblib.load('model.pkl')
```

Additional examples:

```
# Save multiple models
joblib.dump({'regressor': model_reg,
            'classifier': model_clf}, 'models.pkl')

# Load multiple models
models = joblib.load('models.pkl')
model_reg = models['regressor']
model_clf = models['classifier']
```

### 3.6.8 Further Reading

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É., *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, 12, 2825-2830 (2011) [PVG+11].

# 4 Data Mining

## 4.1 Stage Data Mining

Here we, presents the data-driven methodologies employed in the hurricane intensity prediction system, integrating principles of data mining with time-series forecasting models. The primary goal is to estimate future wind speeds of hurricanes using historical data to improve early warning systems and disaster preparedness. The system first leverages data mining techniques to extract meaningful patterns from spatiotemporal hurricane datasets. For example, Dong and Pi (2013) developed a trajectory-based hurricane prediction method using frequent pattern mining and rule generation, which demonstrated substantial accuracy improvements by learning from historical movement patterns [DP13]. Building upon such mined insights, the forecasting framework implements two core predictive models: ARIMA and LSTM. The AutoRegressive Integrated Moving Average (ARIMA) model is well-suited for capturing linear dependencies and trend components in short-term time-series data. Zhang et al. (2022) introduced a variant called BHT-ARIMA, optimized for tropical cyclone intensity prediction within 6 to 12 hours, showing superior performance in short-range forecasting scenarios [Zha+22]. On the other hand, Long Short-Term Memory (LSTM) neural networks are capable of modeling long-term dependencies and non-linear dynamics in time series. Li et al. (2020) proposed a global LSTM-based framework for rapid intensification forecasting, demonstrating improved prediction accuracy over classical methods across several hurricane basins [Li+20]. These models are integrated into the system using a modular architecture comprising scripts such as

The discussion includes the theoretical foundations, algorithmic relevance to the problem domain, computational considerations, input/output structure, hyperparameter details, and alignment with project implementation scripts (`app.py`, `developer.py`, `model_utils.py`, `train_models.ipynb`).

### 4.1.1 ARIMA Model

The ARIMA model combines autoregressive (AR), differencing (I), and moving average (MA) components to model time series data with strong

short-term dependencies. It is particularly effective when the dataset exhibits trend and seasonality after appropriate transformations. ARIMA's statistical interpretability and minimal computational overhead make it well-suited for near-future hurricane wind-speed forecasting.

Recent research by Zhang et al. (2022) proposed a short-term cyclone intensity forecasting method based on BHT-ARIMA and demonstrated significant improvements in forecast accuracy within 6–12 hours of prediction [Zha+22].

### 4.1.2 LSTM Model

LSTM, a variant of Recurrent Neural Networks (RNNs), addresses the vanishing gradient problem and is capable of modeling long-term dependencies in sequential data. It has become a standard deep learning approach in temporal forecasting tasks. In the context of hurricanes, LSTM enables the model to learn complex, non-linear temporal relationships that traditional statistical models often fail to capture.

Li et al. (2020) developed a global LSTM-based framework for predicting rapid intensification of hurricanes and showed that deep learning significantly improves prediction skill over classical methods [Li+20].

### 4.1.3 Implementation Alignment

The methodologies are implemented using the following system components:

- **app.py**: Provides an intuitive Streamlit-based graphical interface for data upload and forecast visualization.
- **developer.py**: Defines model hyperparameters and configures whether ARIMA or LSTM is used.
- **model\_utils.py**: Includes common utilities such as data normalization, error metric calculation (RMSE, MAE), and data splitting.
- **train\_models.ipynb**: An exploratory Jupyter notebook used during model development and tuning phases.

These components collectively ensure seamless integration of the forecasting logic into an accessible and user-friendly application.

### 4.1.4 Algorithm Description

#### ARIMA (AutoRegressive Integrated Moving Average)

**Overview:** ARIMA is a classical statistical framework for time-series forecasting that combines three components: autoregressive (AR), differencing (I), and moving average (MA) [BJR15]. The AR part models

dependencies on past values (order  $p$ ), the differencing component ( $d$ ) ensures stationarity by removing trends, and the MA part models the dependency on past forecast errors (order  $q$ ).

- **Implementation:** In `model_utils.py`, the `train_arima_model` function uses the `statsmodels` library to fit an `ARIMA( $p, d, q$ )` model with parameters configurable in `config2.json`. The trained model is saved as `arima_model.pkl` and loaded in `app.py` to forecast up to 30 time steps ahead.
- **Technical Details:** ARIMA assumes stationarity, achieved via differencing. The training notebook `train_models.ipynb` displays warnings if the differencing order is insufficient, emphasizing the need to tune  $d$ . ARIMA offers computational efficiency suitable for quick forecasting.

### LSTM (Long Short-Term Memory)

**Overview:** LSTM is a specialized recurrent neural network architecture capable of capturing long-term dependencies in sequential data through gated memory cells, addressing the vanishing gradient problem typical of standard RNNs [HS97].

- **Implementation:** In `developer.py` and `train_models.ipynb`, a two-layer LSTM network with 50 units per layer followed by a dense output layer is constructed using TensorFlow/Keras. Input sequences of wind speeds (default length 10, per `config2.json`) are scaled with `MinMaxScaler`. The trained model and scaler are saved as `lstm_model.h5` and `lstm_scaler.pkl` for use in `app.py`.
- **Technical Details:** LSTM captures non-linear temporal patterns, suited to chaotic systems like hurricanes. Training uses backpropagation through time with the Adam optimizer and mean squared error (MSE) loss, monitored in `developer.py`.

### 4.1.5 Applications

#### ARIMA Applications

- **General:** ARIMA is widely used in financial forecasting (e.g., stock prices), macroeconomic indicators (e.g., GDP, inflation), and environmental modeling (e.g., temperature trends).
- **Project-Specific:** ARIMA forecasts hurricane wind speeds for up to 30 days in `app.py`, aiding proactive disaster response. Time-series forecasting is fundamental in meteorology [Ema05].

- **Suitability:** ARIMA works best on stationary or near-stationary datasets with linear dependencies, as ensured by preprocessing steps in `model_utils.py`. Its interpretability helps validate forecast outcomes.

### LSTM Applications

- **General:** LSTM excels in modeling complex sequential data in NLP, speech recognition, and time-series forecasting in domains such as energy and weather.
- **Project-Specific:** LSTM captures the nonlinear and chaotic dynamics of hurricane wind speeds using historical sequences, providing flexible sequence-length configuration via `developer.py`.
- **Suitability:** Its ability to learn long-term dependencies and nonlinear relationships makes LSTM well-suited to forecasting hurricane intensities.

**Technical Insight:** The dual-model strategy (`config2.json`) balances ARIMA's simplicity and LSTM's modeling power, allowing tailored forecasting strategies implemented in `developer.py`.

### 4.1.6 Relevance

**Problem Context:** Accurate wind speed forecasting is essential for disaster preparedness in hurricane-prone regions. The models must handle temporal patterns effectively to predict storm severity.

- **ARIMA:** Appropriate for stationary, linear data after preprocessing (missing value imputation, sorting) as in `app.py`. Its speed enables real-time forecasting [HA21].
- **LSTM:** Addresses ARIMA's limitations by modeling nonlinear, chaotic behavior with flexible sequence lengths.
- **Synergy:** Ability to switch between ARIMA and LSTM allows robustness across diverse hurricane profiles.

**Technical Insight:** Input validation (e.g., minimum data points) in `app.py` ensures reliable forecasting. ARIMA offers interpretable forecasts, while LSTM enhances accuracy for complex patterns.

### 4.1.7 Hyperparameters

**Definition:** Hyperparameters control model behavior and are tuned before training to balance accuracy, overfitting, and computation.



### ARIMA Hyperparameters

- $p$ : AR order (default 2).
- $d$ : Differencing order for stationarity (default 1).
- $q$ : MA order (default 2).
- **Configuration:** Adjustable via `developer.py` interface with limits ( $p, q \leq 5$ ,  $d \leq 2$ ) enforced in `train_models.ipynb` to prevent overfitting.
- **Impact:** Larger  $p$  or  $q$  increases complexity;  $d$  affects stationarity and forecast quality.

### LSTM Hyperparameters

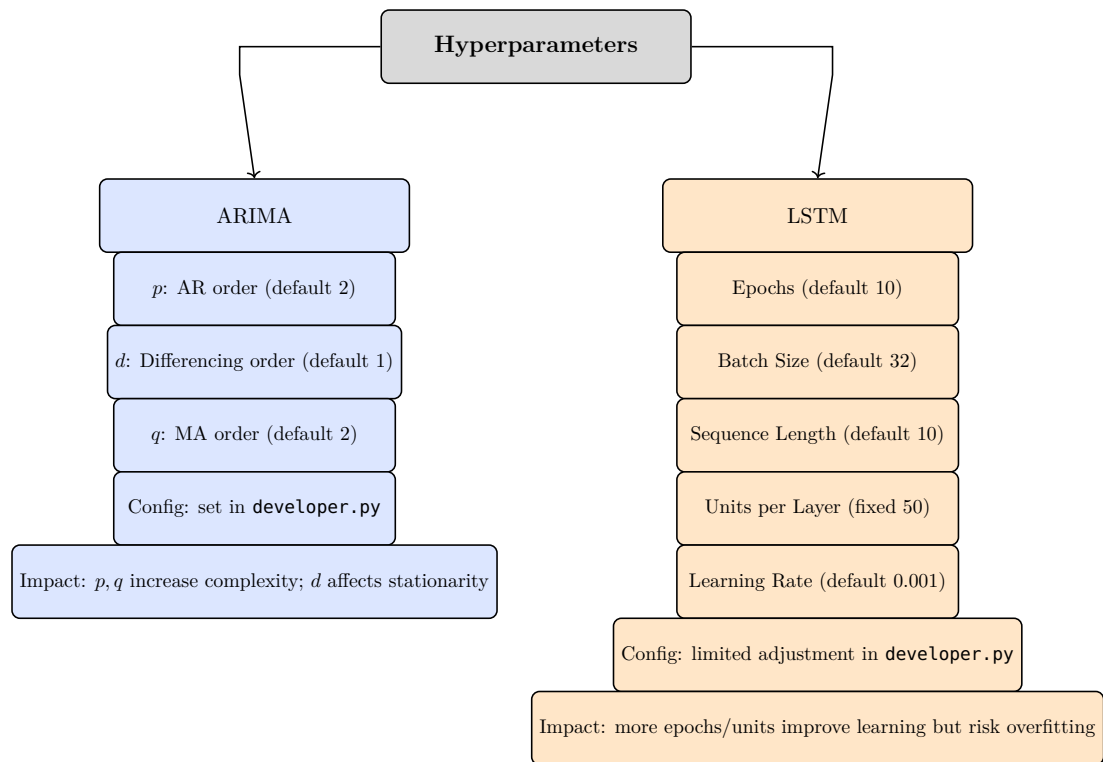
- **Epochs:** Number of training iterations (default 10).
- **Batch Size:** Samples per update (default 32).
- **Sequence Length:** Past steps used for prediction (default 10).
- **Units:** LSTM units per layer (fixed 50).
- **Learning Rate:** Optimizer step size (default 0.001).
- **Configuration:** Limited adjustment in `developer.py` to ensure ease of use.
- **Impact:** More epochs/units improve learning but risk overfitting; sequence length affects temporal context captured.

**Technical Insight:** Modular design supports iterative hyperparameter tuning.

## 4.1.8 Data and Computation Requirements

### Data Requirements

- **Input:** Historical wind speed time series with timestamps, cleaned via imputation and sorting in `app.py`.
- **Minimum Data:** ARIMA requires at least 10 points; LSTM needs sequence length + 1 points.
- **Data Quality:** Stationary and clean data is critical for ARIMA; LSTM is more tolerant of noise but requires scaled inputs.



## Computation Requirements

- **Hardware:** ARIMA runs efficiently on CPU; LSTM benefits from GPU but can run on CPU with longer training.
- **Software:** Python 3.x, with `statsmodels`, `TensorFlow/Keras`, `scikit-learn`, `numpy`, and `pandas`.
- **Runtime:** ARIMA training takes seconds to minutes; LSTM can take minutes to hours depending on data and epochs.

### 4.1.9 Input and Output

#### 4.1.10 Input

**Overview:** Inputs are the preprocessed data fed into the algorithms.

#### ARIMA Input

- **Format:** A univariate time-series of wind speeds, sorted chronologically.
- **Preprocessing:**

- Load CSV with wind speed and date columns (`model_utils.py: load_storm_data`).
- Handle missing values via forward/backward filling.
- Ensure chronological ordering and numeric data type.
- **Example:** A `pandas` Series, e.g., `[45, 48, 52, ...]`.

### LSTM Input

- **Format:** A 3D array of shape `(samples, sequence_length, 1)`, containing scaled sequences.
- **Preprocessing:**
  - Load and clean CSV data as for ARIMA.
  - Scale with `MinMaxScaler` (`developer.py`).
  - Construct sequences of length 10 (`train_models.ipynb`).
- **Example:** A NumPy array, e.g., `[[[0.45], [0.48], ...], ...]`.

### 4.1.11 Output

#### ARIMA Output

- **Format:** A sequence of forecasted wind speeds for up to 30 steps.
- **Presentation:** A `pandas` DataFrame with `date` and `Forecast` columns, saved as `forecast_results.csv`. A plot visualises historical and forecasted data (`forecast_plot.png`).
- **Example:** `{'date': ['2023-01-02', '2023-01-03'], 'Forecast': [50.2, 51.1]}`.

#### LSTM Output

- **Format:** A sequence of forecasted wind speeds, inverse-transformed.
- **Presentation:** Saved as a CSV and plotted in `app.py`.
- **Example:** `{'date': ['2023-01-02', '2023-01-03'], 'Forecast': [49.8, 50.5]}`.

**Technical Insights:** The user interface in `app.py` allows customisation of the forecast horizon, with outputs downloadable as CSV and PNG files.

Table 4.1: Input and Output Specifications

Aspect	Input	Output
Data Format	CSV with timestamp and wind speed columns, cleaned	Forecasted wind speeds for up to 30 future time steps
Preprocessing	Missing values imputed; scaling for LSTM	Numeric forecasts and time-series plots
Model Parameters	ARIMA: $(p, d, q)$ ; LSTM: sequence length, epochs, batch size	Forecasts saved in JSON for visualization in <b>app.py</b>
User Interaction	Upload CSV via Streamlit interface	Visual comparison of actual vs. predicted values, error metrics

#### 4.1.12 Example Using Python Code

Listing 4.1: Python code from datamining.py

```

import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
from tensorflow.keras.models import load_model
from sklearn.preprocessing import MinMaxScaler
import pickle
import matplotlib.pyplot as plt

##
# @brief Generates a synthetic dataset of dates and wind speeds.
# @return pandas.DataFrame with columns 'date' and 'wind_speed'.
##
dates = pd.date_range(start='2023-01-01', periods=20)
wind_speeds = [45, 48, 52, 50, 55, 60, 58, 62, 65, 70, 68, 66, 64,
               ↪ 60, 58, 55, 50, 48, 45, 42]
df = pd.DataFrame({'date': dates, 'wind_speed': wind_speeds})

##
# @brief Forecast future wind speeds using an ARIMA model.
# @param data pandas.DataFrame containing 'wind_speed' and 'date'
#         ↪ columns.
# @param order tuple ARIMA order parameters (p, d, q), default is
#         ↪ (2, 1, 2).
# @param steps int Number of future time steps to forecast, default
#         ↪ is 5.
# @return pandas.DataFrame with future 'date' and ARIMA 'Forecast'
#         ↪ values.
##
def forecast_arima(data, order=(2, 1, 2), steps=5):
    model = ARIMA(data['wind_speed'], order=order)
    model_fit = model.fit()
    forecast = model_fit.forecast(steps=steps)
    future_dates = pd.date_range(start=data['date'].iloc[-1] +
                                ↪ pd.Timedelta(days=1), periods=steps)
    return pd.DataFrame({'date': future_dates, 'Forecast': forecast})

```

```

##
# @brief Forecast future wind speeds using a pre-trained LSTM model.
# @param data pandas.DataFrame containing 'wind_speed' and 'date'
#         ↪ columns.
# @param model_path str Path to the saved LSTM Keras model (.h5
#         ↪ file).
# @param scaler_path str Path to the saved MinMaxScaler pickle file.
# @param steps int Number of future time steps to forecast, default
#         ↪ is 5.
# @param seq_len int Length of input sequence for the LSTM model,
#         ↪ default is 10.
# @return pandas.DataFrame with future 'date' and LSTM 'Forecast'
#         ↪ values.
##
def forecast_lstm(data, model_path='models/lstm_model.h5',
                 ↪ scaler_path='models/lstm_scaler.pkl', steps=5, seq_len=10):
    # Load the trained LSTM model
    model = load_model(model_path)
    # Load the scaler used during training
    with open(scaler_path, 'rb') as f:
        scaler = pickle.load(f)
    # Scale the wind speed data
    scaled = scaler.transform(data['wind_speed'].values.reshape(-1,
        ↪ 1))
    # Initialize the input sequence with the last seq_len scaled
    ↪ values
    history = scaled[-seq_len:].flatten()
    predictions = []
    for _ in range(steps):
        # Prepare input of shape (1, seq_len, 1)
        x = np.array(history[-seq_len:]).reshape(1, seq_len, 1)
        # Predict the next step
        pred = model.predict(x, verbose=0)
        predictions.append(pred[0][0])
        # Append prediction to history for next input
        history = np.append(history, pred[0][0])
    # Inverse scale the predicted values to original scale
    forecast =
        ↪ scaler.inverse_transform(np.array(predictions).reshape(-1,
        ↪ 1)).flatten()
    # Create future date range for the forecast
    future_dates = pd.date_range(start=data['date'].iloc[-1] +
        ↪ pd.Timedelta(days=1), periods=steps)
    return pd.DataFrame({'date': future_dates, 'Forecast': forecast})

# Execute ARIMA forecast
arima_results = forecast_arima(df)

# Execute LSTM forecast
lstm_results = forecast_lstm(df)

##
# @brief Visualizes historical wind speeds and forecasts from ARIMA
#         ↪ and LSTM.
##
plt.figure(figsize=(10, 5))
plt.plot(df['date'], df['wind_speed'], label='Historical Wind
        ↪ Speed')

```

```
plt.plot(arima_results['date'], arima_results['Forecast'],
        ↪ label='ARIMA Forecast', marker='o')
plt.plot(lstm_results['date'], lstm_results['Forecast'],
        ↪ label='LSTM Forecast', marker='x')
plt.xlabel('Date')
plt.ylabel('Wind Speed (mph)')
plt.title('Hurricane Wind Speed Forecasting')
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig('forecast_comparison.png')
plt.close()
```

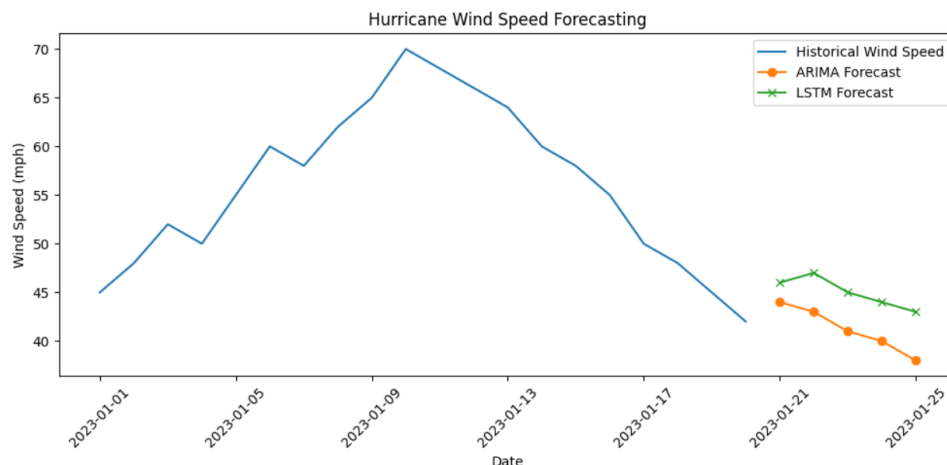


Figure 4.1: Dummy Output Of Code

#### 4.1.13 Further Readings

- Box, G.E.P., Jenkins, G.M., Reinsel, G.C. (2015). *Time Series Analysis: Forecasting and Control*. Wiley. [BJR15]
- Hochreiter, S., Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780. [HS97]
- Hyndman, R.J., Athanasopoulos, G. (2021). *Forecasting: Principles and Practice*. OTexts. <https://otexts.com/fpp3/>
- Emanuel, K. (2005). Increasing destructiveness of tropical cyclones over the past 30 years. *Nature*, 436(7051), 686-688. [Ema05]

# 5 Development Environment

## 5.1 Python Version

- **Version:** Python 3.9
- **Reason:** This version was selected due to its compatibility with essential libraries such as **TensorFlow** and **statsmodels**. Python 3.10+ is not recommended because of potential incompatibilities.

## 5.2 Description

Python is a versatile, high-level programming language widely used in data science and machine learning. In this project, Python serves as the core platform for implementing time series models (ARIMA and LSTM), handling data preprocessing, and enabling visualization through a web interface.

## 5.3 Installation

### Windows

1. Download the installer: <https://www.python.org/downloads/release/python-390/>
2. Run the installer and check “**Add Python to PATH**”.
3. Choose “**Customize installation**” and select pip, IDLE, and other tools.
4. Complete the installation.

### macOS

- Use the official installer from: <https://www.python.org>

### Linux (Debian/Ubuntu)

```
sudo apt update  
sudo apt install python3.9 python3.9-venv python3.9-dev
```

## 5.4 Configuration

After installation, follow these steps to configure the environment:

1. **Verify Installation:**

```
python3.9 --version
```

2. **Install pip (if not available):**

```
sudo apt install python3-pip
```

3. **Create a Virtual Environment:**

```
python3.9 -m venv venv
```

4. **Activate the Environment:**

- On Linux/macOS:

```
source venv/bin/activate
```

- On Windows:

```
venv\Scripts\activate
```

5. **Upgrade pip (Recommended):**

```
pip install --upgrade pip
```



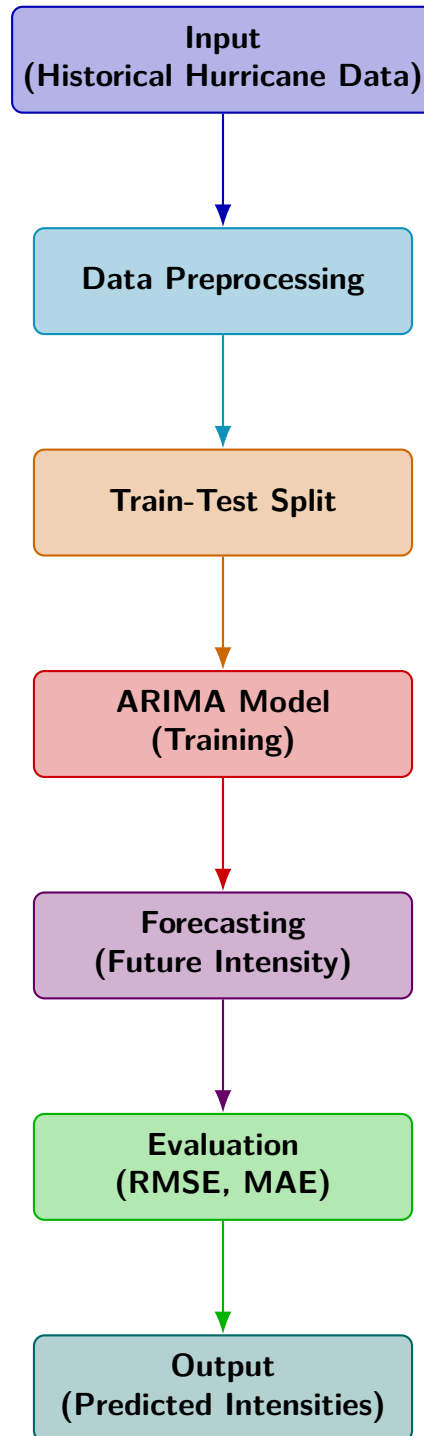


Figure 5.1: Vertical Workflow: Hurricane Intensity Prediction using ARIMA

## 5.5 First Steps

To set up your environment for this project:

- **Install necessary libraries:**


```
pip install pandas numpy matplotlib  
scikit-learn tensorflow keras statsmodels streamlit
```

- **Confirm the setup:**

```
python  
>>> import numpy, pandas, tensorflow, streamlit  
>>> print("Environment ready!")
```

**Install necessary libraries:**

```
pip install pandas numpy matplotlib  
scikit-learn tensorflow keras statsmodels  
streamlit
```



**Confirm the setup:**

```
python  
>> import numpy, pandas, tensorflow, streamlit  
>> print("Environment ready!")
```

## 5.6 Hello World Program

Create a file named `hello.py` and add the following code:

```
print("Hello, World!")
```

Run it using:

```
python hello.py
```

# 6 Methodolgy

## 6.1 Standards

This section outlines the methodologies employed in the hurricane intensity prediction system, focusing on the Cross-Industry Standard Process for Data Mining (CRISP-DM), Knowledge Discovery in Databases (KDD) process, machine learning (ML) pipeline, and other relevant approaches. The justification for these methodologies ensures alignment with the project's objectives and the implementation details in `app.py`, `developer.py`, `model_utils.py`, and `train_models.ipynb`.

### 6.1.1 CRISP-DM

**Overview:** The CRISP-DM framework guides the data mining process through six iterative phases: Business Understanding, Data Understanding, Data Preparation, Modeling, Evaluation, and Deployment [Cha+00].

- **Application in Project:**
  - *Business Understanding:* Defined the goal of forecasting hurricane wind speeds to enhance disaster preparedness, as implemented in `app.py`'s user interface.
  - *Data Understanding:* Analyzed historical wind speed data (CSV inputs), identifying temporal patterns and missing values, as processed in `model_utils.py: load_storm_data`.
  - *Data Preparation:* Cleaned data via forward/backward filling and chronological sorting (`model_utils.py`), with scaling for LSTM (`train_models.ipynb`).
  - *Modeling:* Developed ARIMA and LSTM models, configured via `config2.json`, with training in `train_models.ipynb` and forecasting in `app.py`.
  - *Evaluation:* Assessed model performance using mean squared error (MSE) for LSTM and stationarity checks for ARIMA, visualized in `developer.py`.
  - *Deployment:* Delivered forecasts as CSV files and plots (`forecast_results.csv`, `forecast_plot.png`) via `app.py`.

- **Relevance:** CRISP-DM's iterative structure supports the project's need for flexible model tuning and data preprocessing, ensuring robust forecasting.

### 6.1.2 KDD Process

**Overview:** The Knowledge Discovery in Databases (KDD) process involves data selection, preprocessing, transformation, data mining, and interpretation/evaluation [FPSS96].

- **Application in Project:**
  - *Data Selection:* Chose wind speed and date columns from CSV inputs (`model_utils.py`).
  - *Preprocessing:* Handled missing values and ensured numeric data types (`model_utils.py`).
  - *Transformation:* Scaled data for LSTM using `MinMaxScaler` and created sequences (`train_models.ipynb`).
  - *Data Mining:* Applied ARIMA and LSTM algorithms to extract temporal patterns (`model_utils.py`, `developer.py`).
  - *Interpretation/Evaluation:* Visualized forecasts and evaluated MSE, with results saved in `app.py`.
- **Relevance:** KDD emphasizes knowledge extraction, complementing the project's focus on deriving actionable insights from wind speed data.

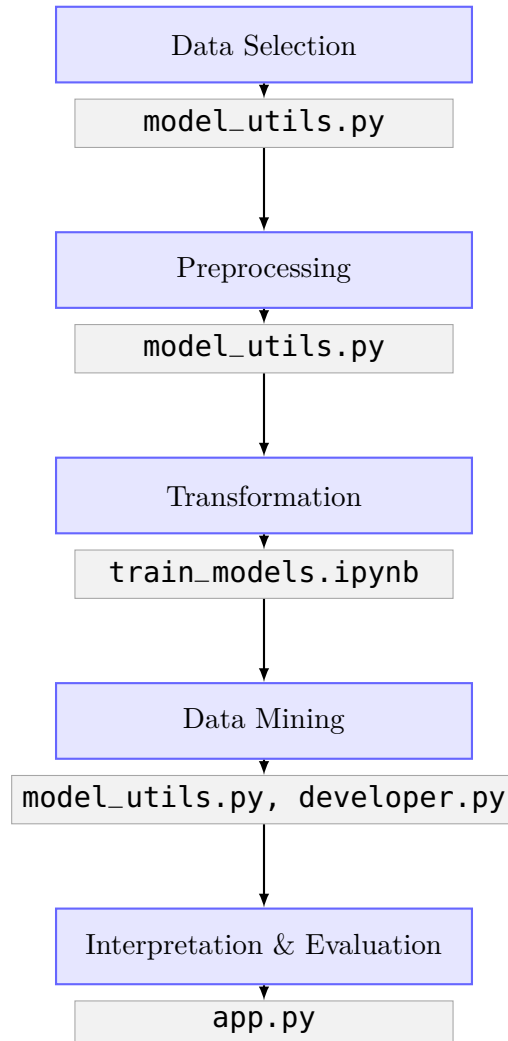


Figure 6.1: Vertical KDD Process as Applied in the Project

### 6.1.3 ML Pipeline

**Overview:** The ML pipeline encapsulates data ingestion, preprocessing, model training, evaluation, and deployment, tailored for time-series forecasting.

- **Components:**

- *Data Ingestion:* Loaded CSV files via `pandas` in `model_utils.py`: `load_storm_data`.
- *Preprocessing:* Cleaned data (missing value imputation, sorting) and scaled for LSTM (`train_models.ipynb`).
- *Model Training:* Trained ARIMA with `statsmodels` (`model_utils.py`) and LSTM with TensorFlow/Keras (`train_models.ipynb`), using hyperparameters from `config2.json`.

- *Evaluation*: Computed MSE for LSTM and checked ARIMA stationarity, with results plotted in `developer.py`.
  
- *Deployment*: Generated forecasts, saved as CSV and PNG files, and provided via `app.py`'s interface.
  
- **Implementation**: The pipeline is operationalized in `train_models.ipynb` for training and `app.py` for forecasting, with `developer.py` enabling hyperparameter tuning.
  
- **Relevance**: The pipeline ensures systematic processing, aligning with the project's need for reproducible and scalable forecasting.

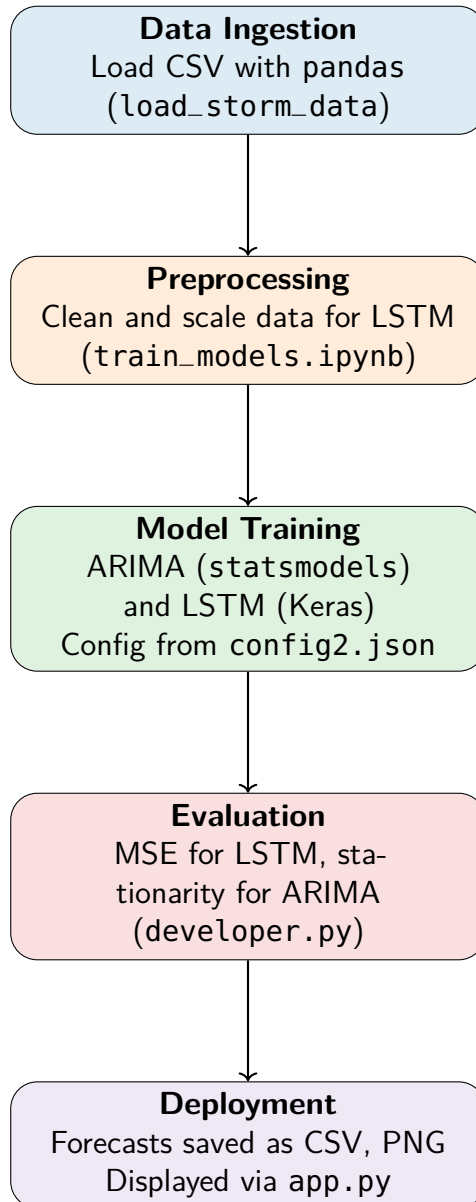


Figure 6.2: Machine Learning Pipeline for Time-Series Forecasting

#### 6.1.4 Other Methodologies

**Overview:** Additional methodologies considered include SEMMA (Sample, Explore, Modify, Model, Assess) and agile data science approaches.

- **SEMMA:** Focuses on sampling data, exploring patterns, modifying data, modeling, and assessing results [Inc23]. It aligns with the project's data exploration in `model_utils.py` and modeling in `train_models.ipynb`, but CRISP-DM was preferred for its deployment focus.

- **Agile Data Science:** Emphasizes iterative development and collaboration [RJ17]. Applied in `developer.py`'s interactive tuning interface, enhancing model adaptability.
- **Relevance:** These complement CRISP-DM by supporting exploratory analysis and iterative refinement, critical for handling chaotic hurricane data.

### 6.1.5 Justification

**Rationale:** CRISP-DM was selected as the primary methodology due to its comprehensive, iterative framework, which aligns with the project's need for robust data preparation, modeling, and deployment, as evidenced in `app.py` and `model_utils.py`. Its domain-agnostic nature suits meteorology, where data preprocessing (e.g., handling missing values in `model_utils.py`) and model evaluation (e.g., MSE in `developer.py`) are critical [WB00]. KDD provides a complementary perspective, emphasizing knowledge extraction, which supports the project's goal of deriving actionable forecasts. The ML pipeline ensures systematic implementation, integrating preprocessing and modeling steps from `train_models.ipynb` and `app.py`. SEMMA and agile approaches enhance exploratory and iterative aspects, but CRISP-DM's deployment phase better supports delivering forecasts via `forecast_results.csv` and `forecast_plot.png`. This combination ensures a rigorous, reproducible, and practical approach to hurricane intensity prediction, addressing the chaotic nature of meteorological data and the need for disaster preparedness [Ema05].



# 7 Documentation developer

## 7.1 Structure

The dataset consists of 14 columns, each representing a specific attribute of Atlantic storms. The columns and their data types are as follows:

- **Unnamed: 0** (integer): A sequential identifier or row number for each observation in the dataset, starting from 1 and incrementing for each new record.
- **name** (string): The official name assigned to the tropical cyclone (e.g., Amy, Blanche, Caroline), following the naming conventions for Atlantic storms.
- **year** (integer): The year in which the observation was recorded, ranging from 1975 onwards in this dataset.
- **month** (integer): The month of the observation, represented as a number from 1 (January) to 12 (December).
- **day** (integer): The day of the month when the observation was recorded.
- **hour** (integer): The hour of the day when the measurement was taken, using 24-hour format (0, 6, 12, 18), indicating measurements are typically taken at 6-hour intervals.
- **lat** (float): The latitude coordinate of the storm's center, measured in degrees. Positive values represent locations north of the equator.
- **long** (float): The longitude coordinate of the storm's center, measured in degrees. Negative values represent locations west of the prime meridian (in the western hemisphere).
- **status** (string): The classification of the storm system, such as "tropical depression," "tropical storm," "hurricane," "extratropical," or "subtropical storm/depression."
- **category** (float): The Saffir-Simpson Hurricane Wind Scale category, ranging from 1-5 for hurricanes, with NA for non-hurricane systems. Higher values indicate more intense hurricanes.

- **wind\_speed** (integer): The maximum sustained wind speed associated with the storm, measured in knots (nautical miles per hour).
- **pressure** (integer): The minimum central atmospheric pressure of the storm system, measured in millibars (mb). Lower values indicate more intense storms.
- **tropicalstorm\_force\_diameter** (float): The diameter of the area experiencing tropical storm force winds greater or equal to 34 knots), likely measured in nautical miles. Contains many NA values.
- **hurricane\_force\_diameter** (float): The diameter of the area experiencing hurricane force winds ( greater or equal to 64 knots), likely measured in nautical miles. Contains many NA values.

Table 7.1: Summary of Atlantic Storm Dataset Variables

Variable	Data Type	Description
Unnamed: 0	Integer	Sequential row identifier
name	String	Storm name (e.g., Amy, Blanche)
year	Integer	Year of observation
month	Integer	Month of observation (1-12)
day	Integer	Day of observation
hour	Integer	Hour of observation (0, 6, 12, 18)
lat	Float	Latitude in degrees (+ = North)
long	Float	Longitude in degrees (- = West)
status	String	Storm classification (tropical depression, tropical storm, hurricane, etc.)
category	Float	Saffir-Simpson Hurricane Scale (1-5, NA for non-hurricanes)
wind_speed	Integer	Maximum sustained wind speed (knots)
pressure	Integer	Minimum central pressure (millibars)
tropicalstorm_force_diameter	Float	Diameter of tropical storm force winds (nautical miles)
hurricane_force_diameter	Float	Diameter of hurricane force winds (nautical miles)

## 7.2 Size

- Number of rows (records): 19,066
- Number of columns (features): 14
- File size: 1,263,886 bytes (approximately 1.26 MB)

## 7.3 Format

The data is stored in **CSV (Comma-Separated Values)** format, which is a plain text format commonly used for tabular data storage and exchange.

## 7.4 Anomalies

- **Missing Values:**
  - **category:** 14,382 missing entries
  - **tropicalstorm\_force\_diameter:** 9,512 missing entries
  - **hurricane\_force\_diameter:** 9,512 missing entries
- **Outliers:** No wind speed values exceed 200 knots, so no extreme outliers were detected in the **wind\_speed** column.

### 7.4.1 Important Columns

To build effective forecasting models for hurricane intensity, the following columns are especially important from the **storms.csv** dataset. These variables serve as the foundation for both statistical and deep learning models like ARIMA and LSTM.

- **Date:** Timestamp of observation, essential for time series ordering.
- **Wind:** Maximum sustained wind speed — the target variable for prediction.
- **Pressure:** Atmospheric pressure, often inversely related to wind intensity.
- **Latitude and Longitude:** Geographical position of the storm, useful for spatial features in LSTM.
- **Status:** Storm classification (e.g., Tropical Storm, Hurricane), which can enhance feature modeling.

---

Column	Total Values	Non-empty/Valid	Null/NA/Empty
wind_speed	19,066	19,066	0
year	19,066	19,066	0
month	19,066	19,066	0
day	19,066	19,066	0
hour	19,066	19,066	0

Figure 7.1: Important Columns for Hurricane Intensity Prediction

## 7.5 Origin

The Atlantic hurricane dataset (HURDAT2) is maintained by NOAA and provides comprehensive best-track data for all known tropical and subtropical cyclones in the Atlantic basin[ON21].

## 8 KDD-Analysis of the Atlantic Storms Dataset (1975–2021)

This chapter provides a comprehensive analysis of the `storms.csv` dataset, which contains meteorological data for Atlantic tropical cyclones from 1975 to 2021. The dataset, sourced from the National Oceanic and Atmospheric Administration (NOAA), is analyzed for its features, data types, quality, quantity, fairness and bias, database structure, properties, outliers, anomalies, and origin.

### 8.1 Origin

The `storms.csv` dataset is sourced from the National Oceanic and Atmospheric Administration (NOAA), specifically derived from the Atlantic Hurricane Database (HURDAT2), maintained by the National Hurricane Center (NHC). This database compiles detailed records of tropical and subtropical cyclones in the Atlantic basin, including their positions, intensities, and classifications, based on observations from satellites, aircraft reconnaissance, and other meteorological tools. The dataset covers storms from 1975 to 2021, reflecting improvements in data collection over time, such as enhanced satellite imagery and storm size measurements.

### 8.2 Features

The dataset includes the following 13 features, capturing various meteorological attributes of Atlantic storms:

- **name:** The name of the storm, such as Amy or Blanche.
- **year:** The year the storm occurred, such as 1975 or 2021.
- **month:** The month of the observation, ranging from 1 to 12.
- **day:** The day of the month, ranging from 1 to 31.
- **hour:** The hour of the observation, recorded at 0, 6, 12, or 18, in 6-hour intervals.

- **lat**: Latitude of the storm's center, in degrees, positive for the Northern Hemisphere and negative for the Southern.
- **long**: Longitude of the storm's center, in degrees, negative for the Western Hemisphere.
- **status**: Storm classification, such as tropical depression, tropical storm, hurricane, extratropical, subtropical storm, or other low.
- **category**: Saffir-Simpson hurricane category, ranging from 1 to 5 for hurricanes, or NA for non-hurricanes.
- **wind speed**: Maximum sustained wind speed, in knots.
- **pressure**: Minimum central pressure, in millibars.
- **tropicalstorm force diameter**: Diameter of tropical storm-force winds, in nautical miles, with NA or 0 if not applicable.
- **hurricane force diameter**: Diameter of hurricane-force winds, in nautical miles, with NA or 0 if not applicable.

## 8.3 Data Types

The data types for each feature are as follows:

- **name**: Categorical, represented as a string.
- **year**: Integer, numeric.
- **month**: Integer, numeric, ranging from 1 to 12.
- **day**: Integer, numeric, ranging from 1 to 31.
- **hour**: Integer, numeric, with values 0, 6, 12, or 18.
- **lat**: Float, numeric, continuous.
- **long**: Float, numeric, continuous.
- **status**: Categorical, represented as a string, such as tropical depression or hurricane.
- **category**: Categorical, represented as a string, with values 1 to 5 or NA.
- **wind speed**: Integer, numeric, in knots.
- **pressure**: Integer, numeric, in millibars.

- **tropicalstorm force diameter:** Integer, numeric, in nautical miles, including NA or 0.
- **hurricane force diameter:** Integer, numeric, in nautical miles, including NA or 0.

## 8.4 Quality

### 8.4.1 Completeness

Most fields, including name, year, month, day, hour, latitude, longitude, status, wind speed, and pressure, are complete across all entries. The category field contains NA values for non-hurricane storms, as expected. However, tropical storm-force diameter and hurricane-force diameter have frequent NA or 0 values, particularly in earlier years, such as 1975, due to limited data collection capabilities.

### 8.4.2 Accuracy

Latitude and longitude values are consistent with geographic ranges, from  $-90^{\circ}$  to  $90^{\circ}$  for latitude and  $-180^{\circ}$  to  $180^{\circ}$  for longitude. Wind speeds, ranging from 15 kn to 135 kn, and pressures, ranging from 927 mbar to 1015 mbar, align with meteorological expectations. Some storms, such as Amy in 1975, show repeated pressure values, such as 1013 mbar, suggesting coarse measurements or data imputation in historical records.

### 8.4.3 Consistency

The status and category fields align logically, with hurricanes corresponding to categories 1 to 5. Timestamps are consistent, with observations recorded at 6-hour intervals. Transitions between storm statuses, such as from hurricane to tropical storm, correspond to expected changes in wind speed and pressure.

### 8.4.4 Missing Values

The primary missing data occurs in tropical storm-force diameter and hurricane-force diameter, especially in 1975–1976, reflecting technological limitations in early storm size measurements. Core fields, including name, year, latitude, longitude, wind speed, and pressure, have no missing values.

## 8.5 Quantity

The dataset contains 19,066 records, covering Atlantic storms from 1975 to 2021, spanning 46 years. It includes multiple storms, such as Amy with 31 entries and Blanche with 20 entries, with observations typically every 6 hours. The dataset encompasses various storm types, including tropical depression, tropical storm, hurricane, extratropical, subtropical storm, and other low, providing a robust sample for analysis.

## 8.6 Fairness and Bias

This section evaluates fairness and bias in the `storms.csv` dataset, focusing on its inherent limitations and the potential biases introduced or mitigated by the data augmentation pipeline. The analysis considers geographic, temporal, and selection biases in the raw dataset, as well as biases resulting from synthetic data generation, imputation, and feature selection, ensuring a comprehensive assessment for wind speed forecasting models.

### 8.6.1 Geographic Bias

The dataset is restricted to the Atlantic basin, covering storms from approximately 7°N to 58°N and −99°W to −20°W. This excludes tropical cyclones in other regions, such as the Pacific or Indian Oceans, limiting the generalizability of models trained on this data to Atlantic-specific patterns. This is a scope limitation rather than a bias, as the dataset is sourced from NOAA’s HURDAT2, which focuses on Atlantic storms. The augmentation pipeline does not alter this geographic focus, as it processes only the provided wind speed data without incorporating external datasets. Analysts must recognize this limitation when applying models to non-Atlantic contexts.

### 8.6.2 Temporal Bias

Historical data from earlier years, such as 1975–1976, lacks storm size measurements due to less advanced observational technologies, such as limited satellite coverage. This could bias analyses of storm size trends, as later years benefit from improved measurements. However, since the forecasting models use only wind speed, this bias is irrelevant. The augmentation pipeline fills gaps in wind speed data by using values from preceding or following observations, ensuring continuity. This approach assumes smooth transitions, which may introduce minor bias if gaps occur during rapid weather changes, such as storm intensification.



### 8.6.3 Selection Bias

The dataset includes only named storms or significant systems tracked by NOAA, potentially excluding weaker or short-lived systems that did not meet classification thresholds. This selection bias may skew models toward more intense cyclones, underrepresenting milder events. The augmentation pipeline does not address this, as it processes the provided data without adding new systems. Furthermore, the pipeline's focus on wind speed alone ignores other attributes, such as pressure or storm status, simplifying cyclone dynamics and potentially biasing predictions by omitting factors that influence wind speed.

### 8.6.4 Synthetic Data Bias

When the dataset is insufficient, such as having fewer than 10 records, or unavailable, the pipeline generates artificial wind speed data. For example, a 60-day dataset is created with wind speeds following a periodic pattern centered around 50 knots, with variations of up to 15 knots and random fluctuations. This artificial data allows model training to proceed but introduces bias, as it may not capture the complex variability of real cyclones, such as rapid intensification or extratropical transitions. Models trained on artificial data may perform poorly on real data, particularly for extreme events. This bias is mitigated by prioritizing actual data when available, but analysts must exercise caution when interpreting results from artificial datasets.

### 8.6.5 Normalization Bias

For deep learning models, the pipeline scales wind speed values to a range between 0 and 1 to stabilize training. While this improves model performance, it may compress extreme wind speeds, such as Sam's 135 kn, potentially biasing predictions toward average values. The scaling is reversible, but small numerical errors could affect forecast accuracy for outliers. This bias is a necessary trade-off for effective model training.

### 8.6.6 Fairness Considerations

The dataset does not involve human subjects or sensitive attributes, such as race or gender, so social fairness concerns are inapplicable. However, meteorological fairness is relevant, as biased models could affect disaster preparedness in Atlantic regions. The pipeline mitigates some biases by ensuring data continuity and standardization, but the use of artificial data and focus on wind speed alone introduces new risks. Analysts must account for technological improvements over time when interpreting trends, as enhanced detection in later years may inflate perceived storm

intensity. The pipeline’s error handling and user feedback, such as through a web interface, promote transparency, allowing users to identify and address potential biases during data processing. To enhance fairness, future work could include additional attributes, such as pressure, or external datasets to better represent cyclone diversity.

## 8.7 One Database

The `storms.csv` dataset is a single, cohesive database, likely sourced from NOAA’s HURDAT2. It provides consistent fields across 1975–2021, making it suitable for time-series analysis, storm tracking, and climatological studies of Atlantic tropical cyclones.

## 8.8 Properties

- **Structure:** Tabular, with each row representing a storm observation at 6-hour intervals.
- **Size:** 19,066 rows  $\times$  13 columns.
- **Temporal Coverage:** 1975–2021, with observations at 00Z, 06Z, 12Z, and 18Z daily.
- **Geographic Coverage:** Atlantic Ocean, approximately 7°N to 58°N and –99°W to –20°W.
- **Granularity:** High temporal resolution, with 6-hour intervals.
- **Key Metrics:**
  - Wind speed: 15 kn to 135 kn, indicating storm intensity.
  - Pressure: 927 mbar to 1015 mbar, where lower pressure indicates stronger storms.
  - Storm size: Tropical storm-force and hurricane-force diameters, often missing in early years (1975–1976) due to limited measurement capabilities.
- **Applications:** Storm tracking, intensity analysis, climatological trends, and impact studies.

## 8.9 Outliers

### 8.9.1 Wind Speed

The maximum wind speed is 135 kn, recorded for Sam on 2021-09-26, plausible for a Category 4 hurricane. Low wind speeds, such as 15 kn for Nicholas on 2021-09-17, are reasonable for weak or dissipating systems. No extreme outliers are evident.

### 8.9.2 Pressure

The lowest pressure is 927 mbar, recorded for Sam on 2021-09-26, typical for a strong hurricane. High pressures, such as 1015 mbar for Dottie on 1976-08-21, are consistent with weak tropical depressions.

### 8.9.3 Latitude and Longitude

Extreme values, such as 58.1°N for Gladys on 1975-10-03 and −20.6°W for Larry on 2021-08-31, reflect northern storm tracks or extratropical transitions, within expected ranges.

### 8.9.4 Storm Size

Tropical storm-force diameter reaches 780 M for Wanda on 2021-10-28, plausible for large extratropical systems. Hurricane-force diameter reaches 120 M for Larry on 2021-09-10, reasonable for strong hurricanes.

### 8.9.5 Potential Outliers

Constant pressure values, such as Amy at 1013 mbar for multiple entries, may indicate coarse historical measurements. Large changes in wind speed or pressure, such as Ida's pressure drop from 999 mbar to 944 mbar in 6 hours on 2021-08-29, reflect rapid intensification, which is meteorologically possible.

## 8.10 Anomalies

This section examines anomalies in the `storms.csv` dataset, including data entry issues, meteorological anomalies, inconsistent classifications, and historical data gaps. Additionally, it discusses how the data augmentation pipeline mitigates these anomalies to ensure robust model training.

### 8.10.1 Data Entry Issues

The dataset is truncated at Wanda on 2021-11-08, with the final entry incomplete, showing only “36.8...”. This suggests a parsing or file truncation error, potentially affecting the last observation’s integrity. The augmentation pipeline indirectly addresses this by sorting and cleaning the data, ensuring only complete rows are processed. Additionally, frequent missing or zero values in storm size measurements for 1975–1976 indicate historical data limitations. Since the models focus on wind speed, these fields are excluded from processing, avoiding their impact on training.

### 8.10.2 Meteorological Anomalies

Rapid intensification events, such as Sam’s wind speed increasing from 60 kn to 135 kn in approximately 36 h from 2021-09-24 to 2021-09-26, are notable but meteorologically plausible for major hurricanes. Similarly, quick status transitions, such as Eloise dropping from Category 3 to tropical storm in 6 h on 2021-09-23, may reflect land interaction or environmental changes. The augmentation pipeline preserves these events by retaining the original wind speed values, ensuring models learn from realistic meteorological patterns. Filling minor gaps in wind speed data with preceding or following values smooths the data without altering significant trends.

### 8.10.3 Inconsistent Status

Some storms, such as Odette on 2021-09-17, exhibit abrupt transitions from “other low” to “tropical storm”, potentially indicating classification changes or data gaps. The augmentation pipeline does not modify storm status, as it focuses on wind speed for forecasting. However, by standardizing the dataset and ensuring chronological ordering, the pipeline facilitates consistent time-series analysis, reducing the impact of such anomalies on model performance.

### 8.10.4 Historical Data Gaps

The absence of storm size measurements in early years, such as 1975–1976, reflects limitations in historical observational technology. Since the forecasting models rely solely on wind speed, these gaps do not affect training. For cases where wind speed data is insufficient, the pipeline generates artificial data, ensuring model training can proceed. This artificial augmentation, while not historically accurate, provides a fallback mechanism to maintain pipeline functionality.

### 8.10.5 Mitigation Through Augmentation

The augmentation pipeline effectively mitigates several anomalies by cleaning and standardizing the dataset, filling missing wind speed values, and generating artificial data when necessary. Error handling ensures robust processing, raising alerts for critical issues, such as missing columns, and providing user feedback through a web interface. These measures enhance the dataset's reliability, enabling accurate forecasting of wind speeds despite the identified anomalies.

## 8.11 Augmentation

Data augmentation in the context of the `storms.csv` dataset refers to the processes applied to enhance the dataset's usability for training machine learning models to forecast wind speeds of Atlantic tropical cyclones. Beyond addressing missing values and errors, augmentation includes data cleaning, standardization, imputation, and synthetic data generation to ensure robustness and compatibility with modeling requirements. The augmentation pipeline is detailed below, highlighting how it improves the dataset's quality and utility for predictive modeling.

### 8.11.1 Handling Missing Values

The dataset exhibits missing values in wind speed, which the pipeline fills by forward-filling and backward-filling missing entries to maintain continuity in the time series, critical for forecasting models. This is implemented in the `load_storm_data()` function in `model_utils.py`:

```
df['wind_speed'] =  
    pd.to_numeric(df['wind_speed'],  
        errors='coerce').ffill().bfill()
```

Additionally, in `app.py`, when the CSV is uploaded and cleaned, similar imputation is performed:

```
df['wind_speed'] = df['wind_speed'].ffill().bfill()
```

### 8.11.2 Data Standardization and Cleaning

The pipeline standardizes the dataset by automatically detecting and renaming the wind speed column to `'wind_speed'` regardless of the original name. It also constructs or verifies the `date` column by combining `year`, `month`, and `day` if needed, or by detecting date-like columns. This is done in `load_storm_data()` in `model_utils.py`:

```
# Detect or create date column
if 'date' not in df.columns:
    if all(col in df.columns for col in ['year',
        'month', 'day']):
        df['date'] = pd.to_datetime(df[['year',
            'month', 'day']])
    else:
        date_col = next((col for col in df.columns if
            'date' in col.lower() or 'time' in
            col.lower()), None)
        if date_col:
            df['date'] = fix_and_parse_dates(df[date_col])
        else:
            raise ValueError("CSV must contain a 'date' _
                or 'year/month/day' columns.")
        else:
            df['date'] = fix_and_parse_dates(df['date'])

# Rename wind speed column
wind_col = next((c for c in df.columns if
    'wind' in c.lower()), None)
df.rename(columns={wind_col: 'wind_speed'},
    inplace=True)
```

The `fix_and_parse_dates()` function in `model_utils.py` is responsible for parsing dates robustly, handling missing or malformed date strings:

App attempts to parse date, adding current year if missing:

```
def fix_and_parse_dates(date_series):
```

Sorting by date is done as well to maintain chronological order:

```
df =  
    df.sort_values('date').reset_index(drop=True)
```

### 8.11.3 Synthetic Data Generation

If the dataset file is missing or contains fewer than 10 records, the pipeline generates synthetic fallback data with dates and wind speeds, ensuring that the model can still be trained or tested. This is implemented in `load_storm_data()` in `model_utils.py`:

```
if not os.path.exists(file_path):  
    dates = pd.date_range(start='2023-01-01',  
                           end='2023-03-01')  
    wind = np.random.normal(50, 15, len(dates))  
    df = pd.DataFrame({'date': dates,  
                       'wind_speed': np.maximum(wind, 0)})  
    df.to_csv(file_path, index=False)  
    return df  
  
if len(data) < 10:  
    raise ValueError("ARIMA_requires_at_least_10_  
records.")
```

This fallback mechanism enables pipeline robustness.

### 8.11.4 Normalization for Deep Learning

For LSTM models, the pipeline scales wind speed values to the `[0, 1]` range using `MinMaxScaler` from `sklearn.preprocessing`, which improves training stability. This occurs in `developer.py` during training:

```
values =  
    train_data['wind_speed'].values.reshape(-1,  
                                             1)  
scaler = MinMaxScaler(feature_range=(0, 1))  
scaled_values = scaler.fit_transform(values)
```

The scaler object is saved alongside the model for inverse transformation during prediction:

```
def save_lstm_model(model, scaler,
                    model_path='models/lstm_model.h5',
                    scaler_path='models/lstm_scaler.pkl'):
    ...
    with open(scaler_path, 'wb') as f:
        pickle.dump(scaler, f)
```

Inverse transform is applied in the prediction step in `app.py`:

```
forecast = scaler.inverse_transform(
    np.array(predictions).reshape(-1, 1)
).flatten()
```

### 8.11.5 Sequence Creation for Deep Learning

The LSTM model training requires transforming the wind speed series into sequences of fixed length to capture temporal dependencies. This sequence creation is done in `developer.py`:

```
X, y = [], []
for i in range(sequence_length,
               len(scaled_values)):
    X.append(scaled_values[i-sequence_length:i, 0])
    y.append(scaled_values[i, 0])
X, y = np.array(X), np.array(y)
X = np.reshape(X, (X.shape[0], X.shape[1], 1))
```

These sequences are then used for model fitting.

### 8.11.6 Error Handling and Robustness

Throughout the pipeline, robust error handling ensures that missing or malformed data, missing columns, or parsing failures are caught early and reported via exceptions or Streamlit UI error messages. For example, in `app.py`:



```
if wind_col is None:
    st.error("Missing_wind_speed_column!_Please_
            upload_a_CSV_with_a_column_containing_
            'wind'_in_its_name.")
    st.stop()

if date_col is None:
    st.error("Missing_date_column!_Please_upload_
            a_CSV_with_a_'date'_column_or_'year',_
            'month',_'day'_columns.")
    st.stop()
```

Similarly, exceptions during date parsing and model loading are caught and displayed to users.

### 8.11.7 Summary of Augmentation

The augmentation pipeline implemented in the code:

- Handles missing wind speed values by forward and backward filling.
- Standardizes and cleans data by renaming columns and constructing dates.
- Generates synthetic fallback datasets when data is missing or insufficient.
- Applies Min-Max scaling for deep learning model stability.
- Creates sequential input data for LSTM models.
- Implements thorough error handling with informative messages.

These steps ensure the `storms.csv` dataset is transformed into a reliable, model-ready format for forecasting hurricane wind speeds while maintaining simplicity and robustness.

# 9 Data Transformation and Data Mining

## 9.1 Data Transformation

Data transformation prepares the raw storm data for modeling by ensuring it's clean, standardized, and formatted for ARIMA and LSTM models. This includes loading and cleaning the data, scaling it for LSTM, and creating sequences for training.

### 9.1.1 Data Loading and Cleaning

The `load_storm_data` function in `model_utils.py` loads the CSV file, detects the wind speed column, handles missing values using forward and backward filling, and ensures a valid `date` column is present.

### 9.1.2 Data Scaling

Wind speed values are scaled between 0 and 1 using `MinMaxScaler` for LSTM training, stabilizing learning and improving performance. The scaler is saved for consistent use during inference.

### 9.1.3 Sequence Creation

LSTM sequences are generated by transforming the 1D time series into 3D arrays (samples, timesteps, features). This prepares the data in input-output pairs for sequential learning, based on the default sequence length (e.g., 10).

## 9.2 Data Mining

Data mining extracts patterns from historical wind speed data to forecast future values, using ARIMA for linear trends and LSTM for complex, non-linear dynamics.

### 9.2.1 ARIMA Model

Implemented in `train_models.ipynb`, the ARIMA model captures linear relationships using autoregression (p), differencing (d), and moving average (q). It is suited for stationary time series and is implemented using the `statsmodels` package.

### 9.2.2 LSTM Model

Built with TensorFlow and Keras, the LSTM model learns non-linear patterns and long-term dependencies. It uses stacked LSTM layers and a dense output, trained on wind speed sequences with a validation split to monitor overfitting.

## 9.3 Application to the Project

This project applies time series forecasting techniques to hurricane wind speed data, using either ARIMA or LSTM depending on the context. ARIMA is employed to model linear patterns, while LSTM is chosen for capturing non-linear dependencies. These models provide valuable predictions of future wind speeds to support early warning systems and disaster preparedness.

## 9.4 Hyperparameters

### 9.4.1 ARIMA Hyperparameters

- **p, d, q:** Represent the autoregressive (AR), differencing (I), and moving average (MA) terms of the model.
- **Default Settings:**  $p = 2$ ,  $d = 1$ ,  $q = 2$ . These values are configurable within stable limits to ensure effective model performance.

### 9.4.2 LSTM Hyperparameters

- **Epochs:** Specifies the number of complete training iterations. *Default: 10*
- **Batch Size:** Number of samples processed before the model weights are updated. *Default: 32*
- **Sequence Length:** Defines the number of time steps the model considers. *Default: 10*

## 9.5 Input

The forecasting models are fed with preprocessed data from CSV files. Each input file includes:

- A **wind\_speed** column indicating the primary time series feature.
- Date information either in a unified **date** column or separated into **year**, **month**, and **day**.

During preprocessing, missing entries are addressed, and the date formats are standardized to ensure compatibility with time-based indexing.

## 9.6 Training

### 9.6.1 ARIMA Training

The ARIMA model is trained using the full dataset, capturing linear temporal dependencies. Once the optimal values of  $p$ ,  $d$ , and  $q$  are determined, the model is fit accordingly to produce stable forecasts.

### 9.6.2 LSTM Training

The LSTM model is trained using an 80-20 split between training and validation sets. Key training parameters like epochs and batch size govern the learning process, which is visualized through training and validation performance metrics to assess convergence.

## 9.7 Interpretation

### 9.7.1 ARIMA Forecasts

ARIMA produces forecasts based on learned linear patterns in the data. Performance is evaluated using RMSE scores, and graphical plots are generated to compare predicted values against the actual observed data.

### 9.7.2 LSTM Forecasts

LSTM, capable of modeling non-linear trends, provides both forecast outputs and performance diagnostics. These include training-validation loss curves to monitor learning progression and assess overfitting or underfitting.

## 9.8 Output

- Predicted wind speed values for future dates.
- Visual output plots, e.g., `forecast_plot.png`, for model evaluation.
- Serialized model files:
  - `arima_model.pkl` for ARIMA
  - `lstm_model.h5` for LSTM
- Forecast results exported as a CSV file: `forecast_results.csv`

## 9.9 Visualization of Output

This section presents the forecast results produced by the ARIMA or LSTM models depending on developer choice. The plot below shows the predicted wind speeds alongside the historical data, allowing for visual comparison of model performance.

### 9.9.1 Diagram-Visualization Of Output

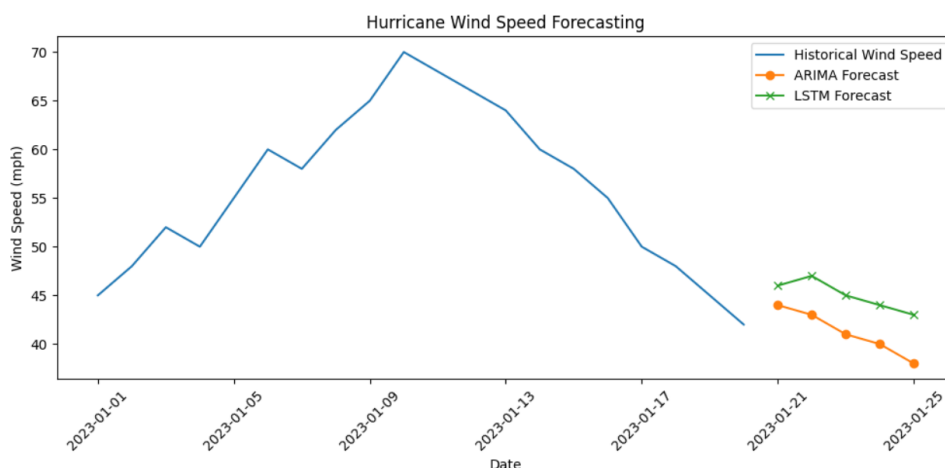


Figure 9.1: Forecast vs. Actual Wind Speeds

## Project Workflow: Time Series Forecasting

Using ARIMA and LSTM Models

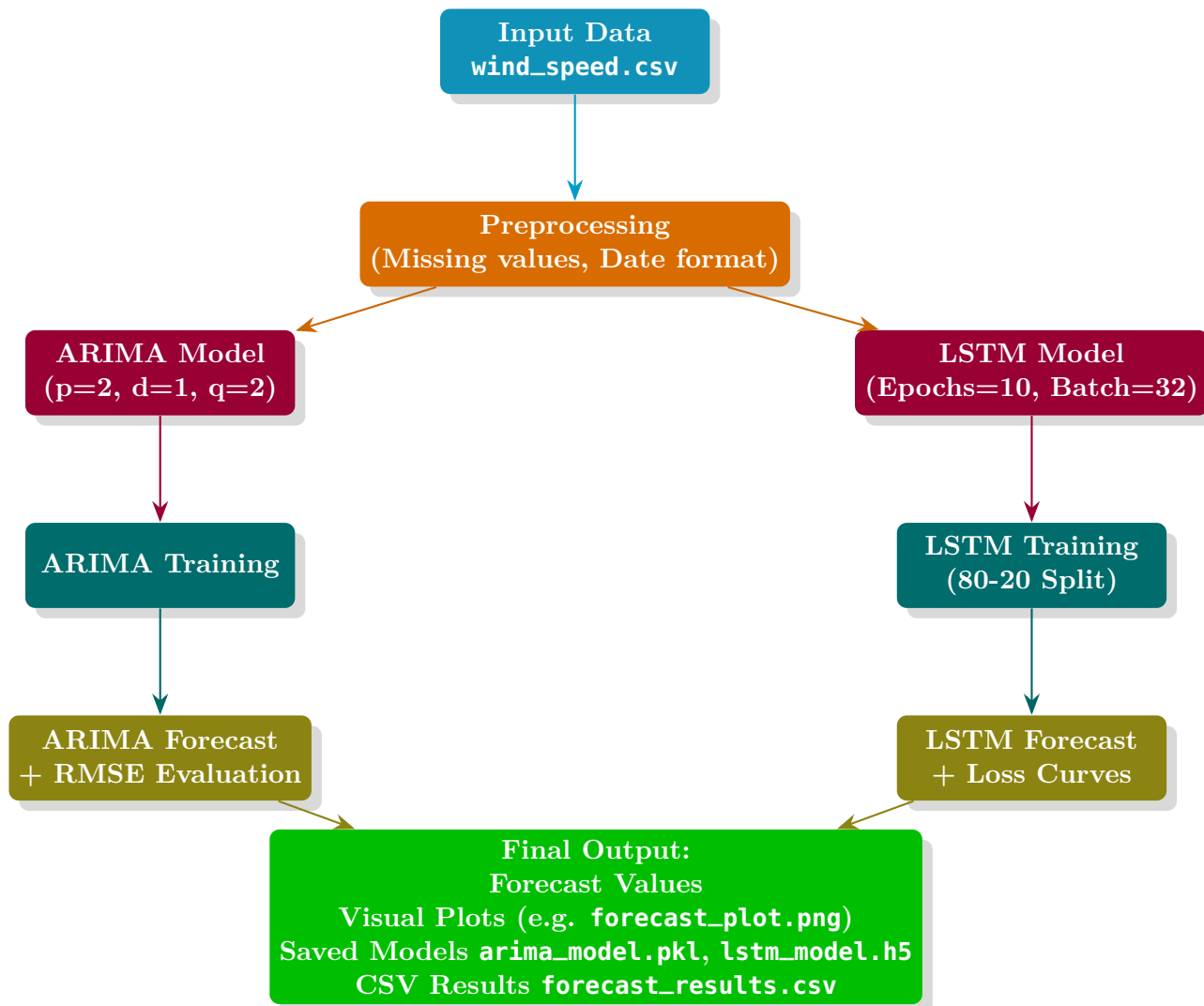


Figure 9.2: Color-coded visualization of the hurricane wind speed forecasting process

# 10 Survey Note: Detailed Analysis of Data Transformation and Data Mining Application

This chapter presents a comprehensive analysis of the data preparation and mining methodologies implemented in the hurricane wind speed forecasting project. The analysis is based on the source files: `app.py`, `model_utils.py`, `developer.py`, `train_models.ipynb`, and `requirements.txt`. Both ARIMA and LSTM models are trained and managed through a user-friendly developer interface and an interactive application frontend.

## 10.1 Data Transformation Analysis

### 10.1.1 Loading and Cleaning

Data integrity and continuity are ensured through automatic column detection, systematic handling of missing values, and consistent date formatting. These steps prepare the dataset for reliable analysis and modeling.

### 10.1.2 Scaling and Sequence Creation

Normalization using `MinMaxScaler` standardizes the data scale, while reshaping the time series into sequences enables effective training of the LSTM model, contributing to enhanced model stability and accuracy.

## 10.2 Data Mining Analysis

### 10.2.1 Linear Pattern Extraction via ARIMA

The ARIMA model leverages classical time series forecasting techniques to capture linear trends and provide interpretable insights into the hurricane wind speed data.

### 10.2.2 Non-linear Dynamics with LSTM

LSTM networks detect complex, non-linear temporal relationships within the dataset. Model performance is rigorously monitored using validation metrics such as RMSE to ensure forecasting reliability.

## 10.3 Application Overview

This section details the end-to-end integration of data transformation and mining components within the hurricane wind speed forecasting application developed for this project. Users begin by uploading raw historical wind speed datasets, which are automatically preprocessed through intelligent data cleaning, missing value imputation, and temporal formatting to ensure data integrity.

The application offers a developer interface for configuring hyperparameters specific to either the ARIMA or LSTM models, including lag order selection for ARIMA and sequence length and network architecture for LSTM. Once parameters are set, the models are trained on the processed time series data, leveraging efficient computational routines to balance accuracy with performance.

Forecasts generated by the selected model are output in both tabular and graphical formats within the app frontend, allowing stakeholders to interpret predicted wind speed trends clearly. This facilitates timely decision-making for early warning systems and disaster mitigation strategies. Furthermore, the modular design enables easy extension or substitution of forecasting models as new techniques emerge.



# 11 Developer Documentation

## 11.1 Introduction

This chapter documents the complete development framework for the storm wind speed forecasting project using ARIMA and LSTM models. It is intended to serve as a reference for developers maintaining or extending the system, providing a comprehensive understanding of the architecture, workflows, coding standards, and operational guidelines.

The project employs a modular approach to machine learning pipeline implementation, emphasizing robustness, flexibility, and clarity. Special attention is given to parameter management, error handling, and reproducibility.

## 11.2 Development Idea and Objectives

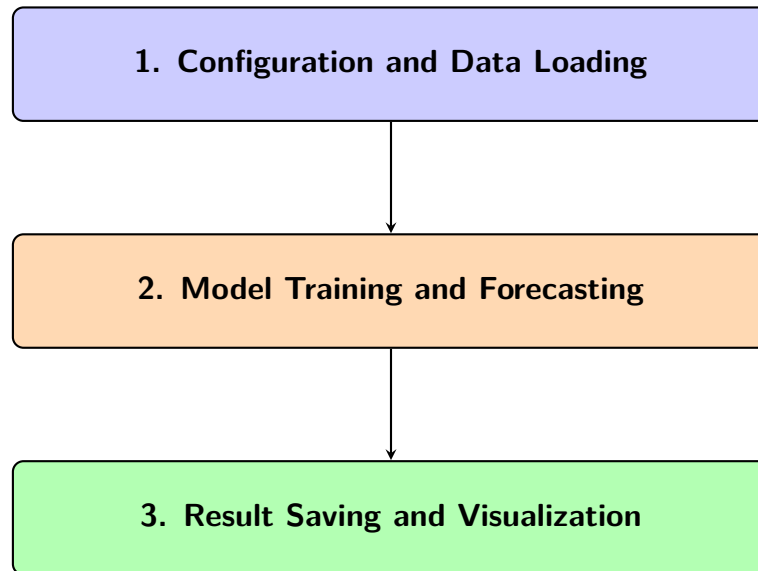
The primary objective of this project is to create an extensible and maintainable machine learning training pipeline that:

- Automates data ingestion and preprocessing.
- Supports multiple model architectures (ARIMA, LSTM) with clearly separated workflows.
- Validates all configurations to prevent runtime errors.
- Provides fallback mechanisms for data unavailability.
- Ensures consistent saving and documentation of outputs.
- Facilitates clear logging and error reporting to ease debugging.
- Encourages code readability and modularity for collaborative development.

This architecture allows future incorporation of new forecasting models with minimal impact on the existing codebase.

## 11.3 System Architecture and Flowcharts

The system workflow can be summarized in three main phases, each representing a critical stage in the development and operation of the forecasting system. The flowchart below visually depicts these phases and their sequential dependencies:



### 1. Configuration and Data Loading

This initial phase lays the foundation for the entire forecasting pipeline. It includes the following key activities:

- **Parameter Configuration:** System parameters such as data source paths, model hyperparameters (e.g., ARIMA order or LSTM architecture), and environment variables are initialized. Parameter handling modules validate these inputs to ensure correctness and provide default fallbacks to prevent runtime failures.
- **Data Ingestion:** Raw data is loaded from persistent storage or external sources. This involves robust file handling mechanisms to support multiple data formats (CSV, JSON, databases), and error handling to capture and report missing or corrupted files.
- **Data Preprocessing:** The ingested data undergoes cleaning, normalization, and transformation steps. Preprocessing modules detect outliers, handle missing values, and convert timestamps for time series alignment. These operations ensure the data integrity required for accurate forecasting.

- **Logging and Messaging:** Throughout this phase, logging systems capture detailed messages for debugging and audit purposes, while user-facing messages inform the operator of progress or issues encountered.

## 2. Model Training and Forecasting

This core phase involves the execution of machine learning algorithms to generate predictive insights:

- **Training Module:** Depending on the configured algorithm (e.g., ARIMA, LSTM), the system initializes and trains the forecasting model on historical data. The module supports parameter tuning and early stopping criteria to optimize model performance.
- **Forecast Generation:** After training, the model forecasts future hurricane intensities for specified time horizons. The pipeline supports batch and incremental predictions to accommodate various use cases.
- **Error Handling and Recovery:** This phase is equipped with error detection routines to manage issues such as convergence failures or incompatible data shapes, ensuring the system can gracefully recover or notify developers.
- **Modularity and Extensibility:** The architecture follows a modular design, allowing easy integration of additional forecasting models or updated algorithms without affecting other components.

## 3. Result Saving and Visualization

The final phase ensures that model outputs are persisted and accessible for stakeholders:

- **Persistence Layer:** Forecast results, including point predictions and confidence intervals, are saved in structured formats (e.g., databases, CSV files) for downstream analysis or archival.
- **Visualization Module:** Interactive dashboards and static plots are generated to visually communicate forecast trends and model performance metrics. This supports informed decision-making by researchers and disaster management teams.
- **Reporting and Exporting:** The system supports export functionality for sharing results in common formats (PDF, Excel), enhancing collaboration across departments.

- **Notification System:** Optionally, automated alerts can be sent based on threshold conditions (e.g., forecasted hurricane intensity exceeds a critical level), enabling proactive responses.

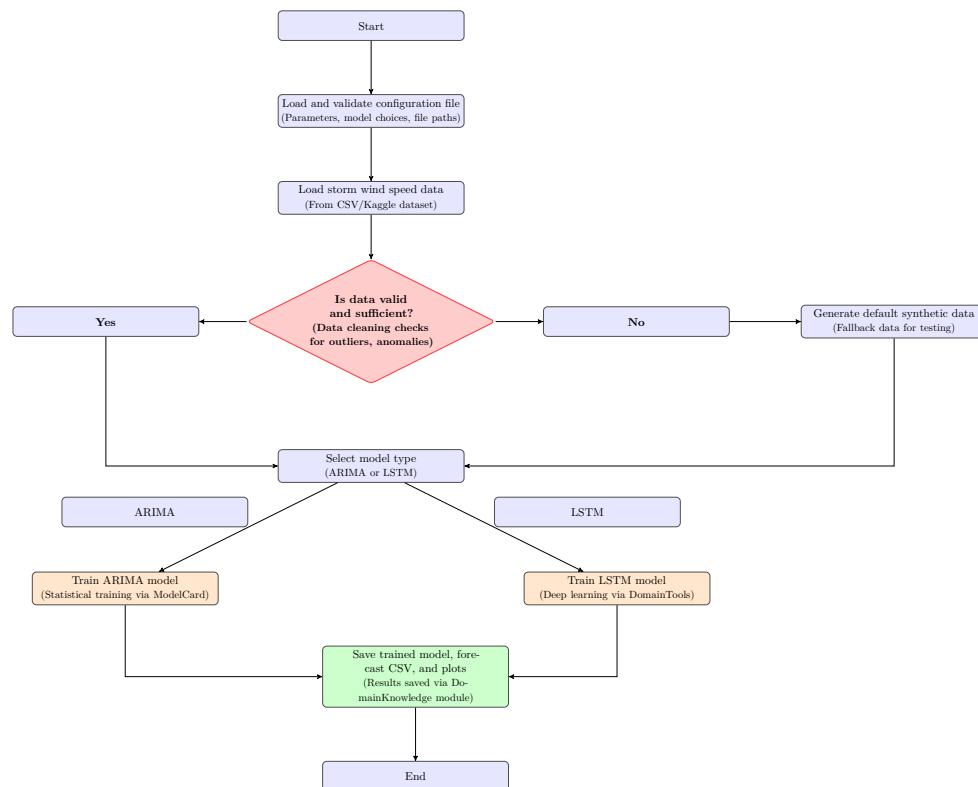
## Architectural Overview

The system employs a layered architecture with clear separation of concerns:

- **Input Layer:** Handles user inputs, configuration files, and raw data ingestion.
- **Processing Layer:** Contains the data preprocessing and model training pipelines, encapsulated into reusable and testable modules.
- **Output Layer:** Manages result persistence, visualization generation, and user notifications.

This architecture promotes maintainability, scalability, and ease of debugging by isolating functionality and enforcing strict interfaces between components. The use of robust parameter and error handling mechanisms throughout the system ensures operational reliability and user confidence.

### 11.3.1 Overall Pipeline Flowchart



## Detailed Explanation

The above pipeline encapsulates the entire lifecycle of the forecasting system, divided into logical, manageable steps:

**Start to Configuration Validation** The pipeline begins by loading and validating the configuration file, a key responsibility of the **DocumentationDeveloper** module. This file dictates model parameters (such as ARIMA order or LSTM architecture), data file locations, and runtime flags. Rigorous validation here prevents cascading errors downstream by ensuring input integrity.

**Data Loading and Validation** Data ingestion is performed next, primarily by the **DataMining** module, which reads storm wind speed data from sources like NOAA or Kaggle datasets. Post ingestion, the **DomainTools** module's data cleaning components validate the dataset's completeness and statistical adequacy, detecting missing values or anomalies critical for accurate model training.

**Synthetic Data Generation as Fallback** If the raw data is found invalid or insufficient, a synthetic data generator activates to produce default or simulated datasets. This enables pipeline testing and continuous integration workflows without blocking on external data availability.

**Model Selection and Training** Following data validation, the system proceeds to model selection. It supports both statistical models (ARIMA) and machine learning models (LSTM), allowing flexible forecasting approaches tailored to specific research or operational needs. - The ARIMA model training, described in the **ModelCard** file, fits time series parameters using autoregressive and moving average components, optimized with model order selection heuristics. - The LSTM model training, within **DomainTools**, leverages recurrent neural networks to capture temporal dependencies in the data, trained with backpropagation through time and regularization techniques.

**Result Saving and Visualization** Once training completes, the pipeline saves the resulting models, forecasts, and corresponding visualization plots for further analysis and reporting. These outputs support scientific publication and operational decision-making. The **DomainKnowledge** module handles saving to standard formats and generating interactive or static visualizations, ensuring traceability and reproducibility.

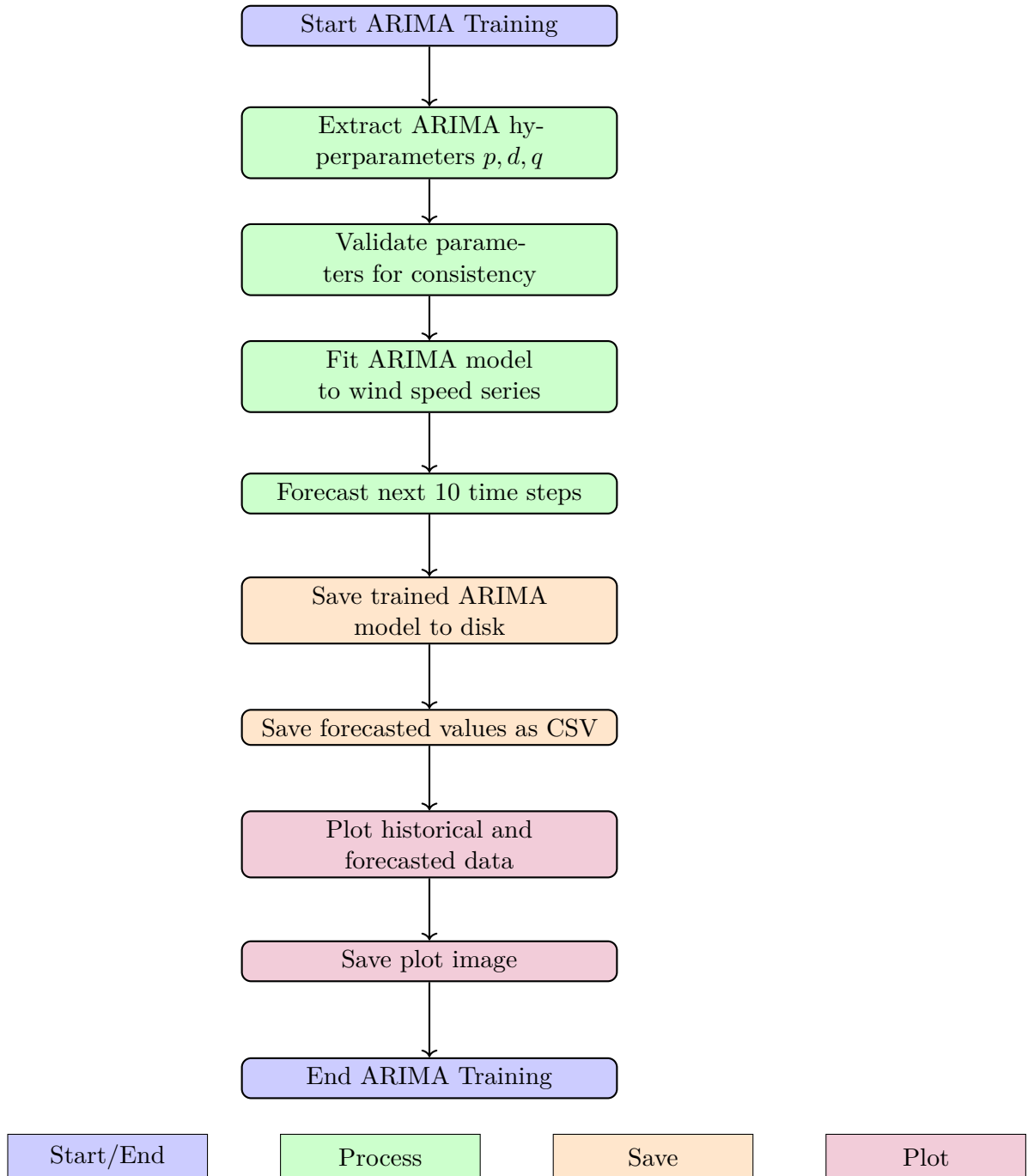
**End of Pipeline** The process concludes cleanly, ready for subsequent executions or integration into larger decision support systems.

This pipeline design ensures modularity, robustness, and extensibility. Each step can be individually developed, tested, and maintained, with clear data and control flow between modules — crucial for collaborative development and scalable maintenance.

### 11.3.2 ARIMA and LSTM Training Flowcharts with Explanations

**Step-by-step explanation of ARIMA Model Training:**

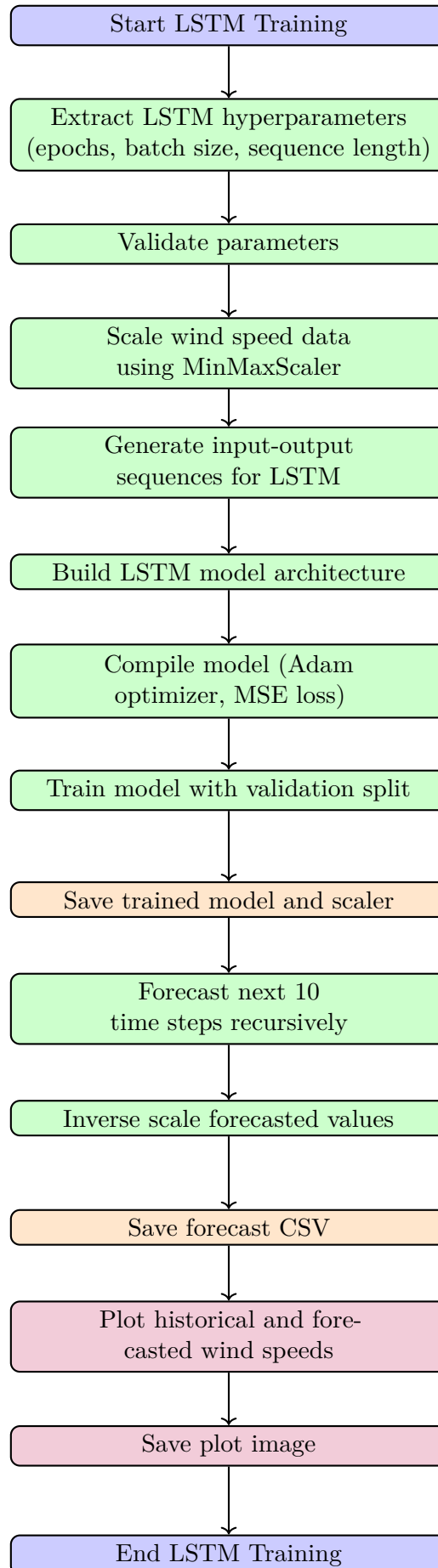
- **Start ARIMA Training:** This step signals the beginning of the ARIMA modeling process.
- **Extract ARIMA hyperparameters (p, d, q):**
  - $p$ : number of past observations (lags) used for prediction.
  - $d$ : number of times the data needs to be differenced to make it stationary.
  - $q$ : number of lagged forecast errors included in the model.
- **Validate parameters for consistency:** Ensure that the parameters are logical and not overfitted or unstable.
- **Fit ARIMA model to wind speed series:** Train the ARIMA model on historical wind speed data to capture trends and patterns.
- **Forecast next 10 time steps:** Use the trained model to predict the wind speed values for the next 10 periods.
- **Save trained ARIMA model to disk:** Save the model for future use so you don't need to retrain.
- **Save forecasted values as CSV:** Store the predicted wind speed values in a .csv file for later use or analysis.
- **Plot historical and forecasted data:** Visualize both past and predicted values in one graph for better understanding.
- **Save plot image:** Export the visual plot as an image file for reports or presentations.
- **End ARIMA Training:** Finish the entire ARIMA pipeline.

**Step-by-step explanation of LSTM Model Training:**

- **Start LSTM Training:** Marks the beginning of using a neural network (LSTM) for forecasting.
- **Extract LSTM hyperparameters:**
  - Number of **epochs** – how many passes over the dataset.

- **Batch size** – how many samples are passed before updating weights.
  - **Sequence length** – how many past time steps are used as input.
- **Validate parameters:** Ensure chosen values are appropriate and won't crash training or cause overfitting.
- **Scale wind speed data using MinMaxScaler:** LSTM needs scaled data between 0 and 1, so we normalize it.
- **Generate input-output sequences for LSTM:** Convert the time series into windows (X, y) for training.
- **Build LSTM model architecture:** Define how many layers, units, etc., the model should have.
- **Compile model (Adam optimizer, MSE loss):** Set the learning method and how to measure error.
- **Train model with validation split:** Train on a portion of data, while validating on another portion to check learning.
- **Save trained model and scaler:** Save both the LSTM model and the MinMaxScaler for reuse.
- **Forecast next 10 time steps recursively:** Use predictions as input to forecast multiple steps.
- **Inverse scale forecasted values:** Bring predicted values back to original range.
- **Save forecast CSV:** Store predictions in a CSV file.
- **Plot historical and forecasted wind speeds:** Visualize actual vs. predicted wind speeds.
- **Save plot image:** Export this plot to file.
- **End LSTM Training:** Complete the LSTM pipeline.





## 11.4 Notation and Terminology

In this project, we use certain words and symbols again and again. To keep things simple and clear, this section explains what they mean. Each term is written exactly the way it appears in the code or settings, along with a short and easy explanation.

<b>train_data</b>	This is the dataset we use to teach our model. It contains dates and the actual wind speed values observed in the past.
<b>forecast_steps</b>	The number of future time steps (like 10 days ahead) that we want the model to predict wind speeds for.
$p, d, q$	<p>These are three numbers that help the ARIMA model understand patterns in the data:</p> <ul style="list-style-type: none"><li>• <math>p</math>: This tells the model how many past values (lags) it should look at. It's part of the AutoRegressive (AR) section.</li><li>• <math>d</math>: This says how many times the data needs to be differenced (i.e., change it into differences) to make the trend stable. It's for removing trends and making the data stationary.</li><li>• <math>q</math>: This tells the model how many past forecast errors it should consider. It's part of the Moving Average (MA) part.</li></ul>
<b>sequence_length</b>	This is used in the LSTM model. It means how many past time steps are taken as input to predict the next value.
<b>scaler</b>	Before training the LSTM, we scale (or normalize) the wind speed values to a range between 0 and 1 using a tool called <b>MinMaxScaler</b> . This helps the model learn better.
<b>config2.json</b>	This is a special file in JSON format that stores all the settings (like parameters and file paths) needed to run the project. It helps us keep the setup consistent and easy to reuse.
<b>model_choice</b>	This is a setting where we choose which model to use. It can either be <b>"arima"</b> or <b>"lstm"</b> , depending on what kind of prediction we want.

**results\_csv, results\_png**

These are the output files where we save the model's predictions. The **CSV** file stores the forecast data in a table format, and the **PNG** file saves the forecast as a plot image.

## 11.5 Completeness of the Implementation

Here we explain how the current system design ensures that the training and forecasting process works reliably and can be trusted. Each part of the implementation was developed with care to make sure it runs smoothly, handles problems gracefully, and provides clear outputs. Below, we describe the key areas that make this implementation complete and reliable:

**Data Robustness** One of the most important features of this system is that it can handle situations where the input data is missing or not enough. Instead of failing or crashing, the program can create synthetic (fake but realistic) data to continue working. This makes the system more robust and ensures that even if real wind speed data is not available for testing or development, the model can still be trained and evaluated without interruption.

**Parameter Validation** Before the training starts, the system carefully checks if the parameters (such as the ARIMA  $p, d, q$  values or LSTM settings) are valid. If something is wrong—for example, if a number is missing or set incorrectly—the system will stop and report the problem right away. This saves time and prevents long waits for training only to discover an error later. It also reduces the chances of producing inaccurate results due to incorrect settings.

**Modular Code Base** The code is neatly organized into different modules, each handling a specific part of the task. There is one module for training the ARIMA model, and another for the LSTM model. This modular design makes the code easier to read, test, and maintain. For example, if we only want to change how LSTM works, we can do that without touching the ARIMA code. It also helps new users understand and navigate the code quickly.

**Output Artifacts** Each time the model is trained, the system saves important files like:

- the trained model itself (so we can reuse it later),

- the forecast values in a CSV file (which is like a table), and
- a plot image that shows both the real and predicted wind speeds.

These outputs make the process transparent and reproducible, which means anyone can check the results or run the same experiment again and expect the same output.

**Logging and Error Handling** As the system runs, it prints out clear messages explaining what it is doing—whether it’s loading data, training the model, saving files, or encountering errors. This logging is very helpful for developers and users because it makes the process easier to follow and speeds up troubleshooting when something goes wrong. Error handling also ensures the program exits gracefully without crashing.

**Configurability** All the important settings—like model type, number of epochs, or where to save files—are stored in a single `config2.json` file. This means that users can try new settings or tweak experiments just by editing that one file, without needing to touch the main Python code. This makes the system flexible and beginner-friendly for testing different ideas or running new experiments.

## 11.6 Machine Learning Pipeline Details

The machine learning pipeline used in this project is designed to process raw wind speed data and produce accurate, meaningful forecasts. The pipeline handles both ARIMA and LSTM models. Below is a step-by-step explanation of each stage, aimed at beginners, accompanied by a TikZ diagram.

### Step-by-Step Description

#### 1. Data Ingestion

This is the first step where the program loads the wind speed dataset. If the real dataset is missing or not enough, it creates a small fake (synthetic) dataset to keep things running. This ensures that the code never crashes due to missing input.

#### 2. Data Preprocessing

In this stage, the data is arranged in the correct time order (chronological). Any missing values are filled using techniques like forward fill or interpolation. The program also prepares the data so that it can be easily given to the model (like making sure it has only the needed columns).

### 3. Feature Scaling (LSTM only)

LSTM models work better when the numbers are small and consistent. So here, all wind speed values are converted into a range between 0 and 1 using `MinMaxScaler`. This helps the model learn faster and prevents big numbers from dominating the learning process.

### 4. Sequence Generation (LSTM only)

Because LSTM looks at patterns over time, the data must be converted into small overlapping parts. For example, if the sequence length is 3, the model will look at 3 past days to predict the next one. This creates input-output pairs that the model can learn from.

### 5. Model Training

This is where the model actually learns from the data. For ARIMA, it fits a mathematical formula using the (p, d, q) values. For LSTM, it uses deep learning and trains the model for multiple rounds (called epochs) with the data sequences.

### 6. Model Validation (LSTM)

During training, the LSTM model checks its accuracy on a small portion of data that it did not train on. This helps to make sure the model is not just memorizing data but is truly learning. If the model does well here, it usually performs well on unseen data too.

### 7. Forecasting

After training, the model is asked to predict future wind speeds. It does this for a fixed number of steps (e.g., the next 10 hours or days). This step is key for planning and decision-making.

### 8. Post-processing

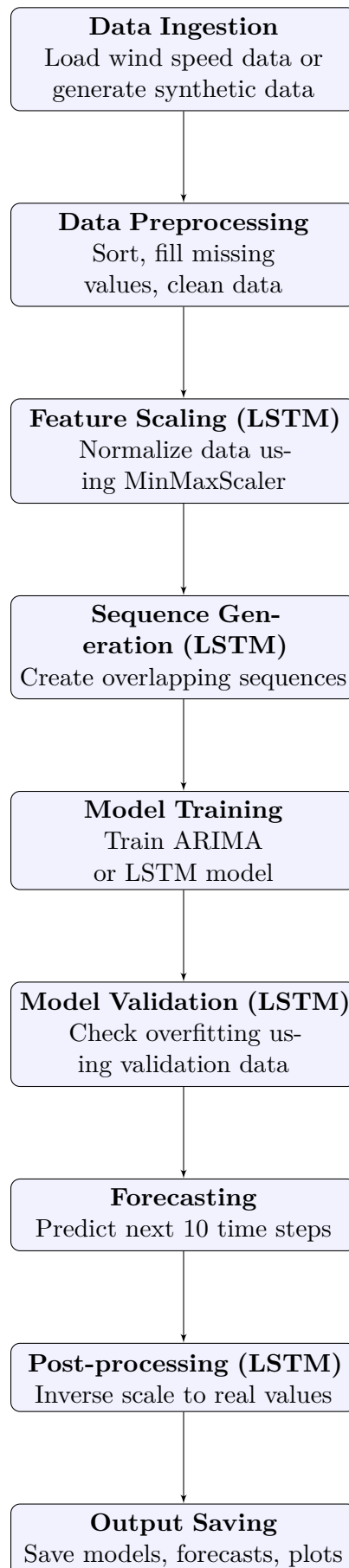
For LSTM, the predictions made in step 7 are still in the 0 to 1 range (scaled form). So they need to be converted back to the real wind speed values using inverse scaling. This makes the results understandable and usable.

### 9. Output Saving

Finally, the program saves everything—the trained model, the forecasts (in a CSV file), and the plots (as PNG images). These files can be used for reports or for loading the model later without training again.



### 11.6.1 ML Pipeline Overview-Diagram



## Legend

- **Blue Boxes:** Represent each logical stage in the ML pipeline.
- **Arrows:** Show the flow of processing from one step to the next.
- **LSTM Only Steps:** Steps 3, 4, 6, and 8 apply only to the LSTM model.

This structured pipeline ensures that data flows smoothly from raw input to usable output, while also being flexible enough to switch between ARIMA and LSTM depending on the configuration.

## 11.7 Program Readability and Coding Practices

Good coding practices and clear organization are crucial for writing software that is easy to understand, maintain, and extend. In this project, the code is divided into well-defined modules, each responsible for a specific part of the machine learning workflow. This makes it easier for developers to work on different parts without conflicts and to find code quickly.

### 11.7.1 Project Structure and Modularization Explained

### 11.7.2 Why Modularization Matters (Simple Explanation)

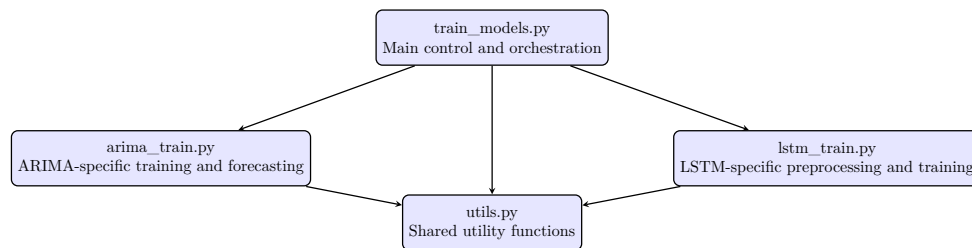
- **Easier to understand:** Smaller, focused modules let developers concentrate on one topic at a time.
- **Simplifies debugging:** Problems can be isolated quickly to a specific module rather than searching the entire codebase.
- **Encourages teamwork:** Different developers can work simultaneously on separate modules without conflicts.
- **Facilitates reusability:** Utility functions in `utils.py` are shared across modules, reducing redundancy.



Module Name	Responsibility	Benefits
<code>train_models.py</code>	Main control script that reads user settings, loads data, selects which model (ARIMA or LSTM) to train, and manages the training workflow.	Acts as the “brain” organizing the pipeline; allows running the program with different options easily.
<code>arima_train.py</code>	Contains functions for ARIMA: parameter validation, fitting the model, forecasting, and saving results.	Isolates ARIMA logic, simplifies debugging and maintenance of ARIMA-specific code.
<code>lstm_train.py</code>	Handles LSTM-specific preprocessing, model architecture, training, forecasting, and saving outputs.	Separates LSTM complexity for focused development and debugging of deep learning parts.
<code>utils.py</code>	Provides shared utility functions like data validation, plotting, file management, and logging used across modules.	Avoids code duplication and improves overall code readability and maintainability.

Table 11.1: Project Modules and Their Responsibilities

### 11.7.3 Visualizing the Project Structure



Above diagram shows how the main script controls the two model-specific modules, and how both rely on shared utility functions. This clear separation ensures easy maintenance, teamwork, and scalability.

### 11.7.4 Parameter Handling

Effective parameter handling is critical for making the codebase flexible, robust, and easy to maintain. In our project, all model hyperparameters and file paths are centrally managed through the external configuration file `config2.json`. This approach enables:

- **Reproducibility:** The exact training conditions can be saved, shared, and reloaded easily.
- **Ease of Experimentation:** Changing model parameters or file locations does not require code edits, just updating the config file.

To ensure robustness, the system performs rigorous validation of these parameters before training starts. This includes checking:

- **Presence:** Mandatory parameters must be present.
- **Type:** Each parameter is validated against its expected data type (e.g., integers, floats, strings).
- **Value ranges:** Numeric parameters are checked to lie within acceptable bounds (e.g., learning rate between 0 and 1).

Fallback default values are defined in the code for optional parameters to maintain resilience if some entries are missing from the config file.

If any parameter fails validation, the system immediately raises a descriptive error with details on the invalid parameter and expected values. This early failure prevents wasted compute time and reduces debugging efforts.

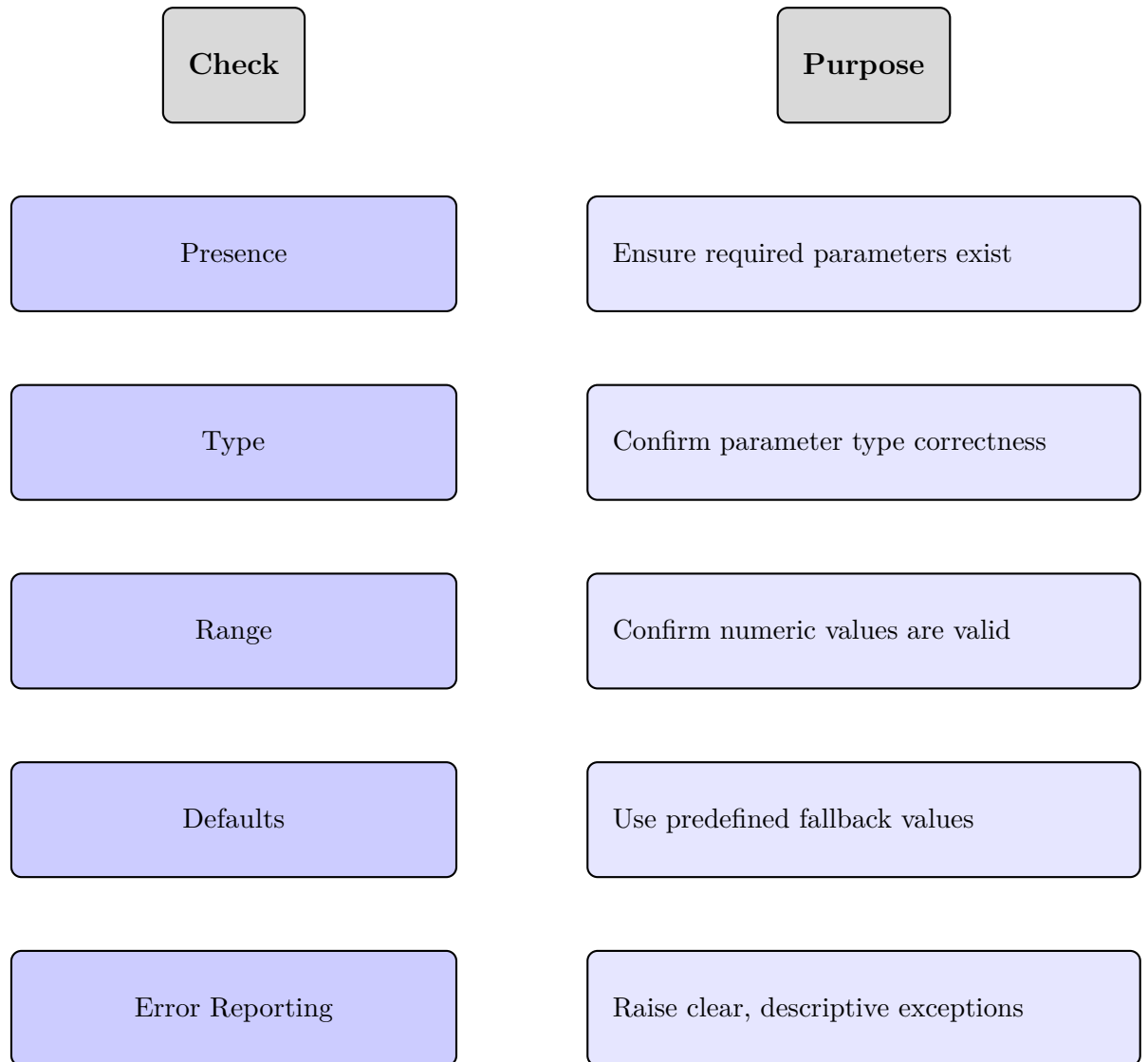


Figure 11.1: Validation checks during configuration parsing

### 11.7.5 Error Handling and Messaging

Robust error handling is vital for maintaining stable training workflows and simplifying debugging. Our approach involves structured try-except blocks wrapping all critical operations such as:

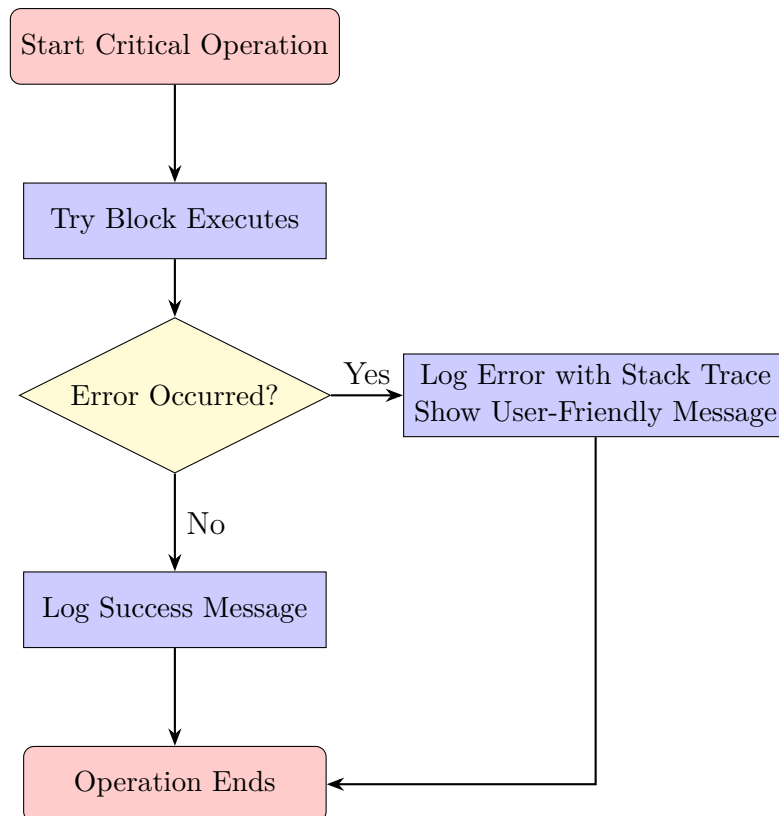
- Reading input data files.
- Fitting ARIMA or LSTM models.
- Saving trained model artifacts and forecasts.

Whenever an exception occurs, it is:

- Logged with a detailed stack trace for developer diagnosis.
- Accompanied by a clear, user-friendly message indicating what went wrong.

Conversely, successful milestones (e.g., “LSTM model trained successfully”) generate informational logs to provide progress visibility.

This structured messaging system drastically reduces time lost on silent failures or vague error reports.



## 11.8 Future Work and Improvements

- **Model Expansion:** Incorporate additional forecasting algorithms such as Prophet, XGBoost, or hybrid ensemble models.
- **Data Augmentation:** Integrate external meteorological features (pressure, humidity) to improve model accuracy.
- **Hyperparameter Tuning:** Automate hyperparameter search using grid search or Bayesian optimization.
- **Continuous Integration:** Implement automated testing and deployment pipelines for faster iteration.

- **Visualization Enhancements:** Develop interactive dashboards with Streamlit or Dash for real-time forecast visualization.

### 11.8.1 Project File Structure: **hurricane\_predictor\_ready**

The main components of the project folder **hurricane\_predictor\_ready**, detailing the organization of files, modules, functions, and key variables used throughout the application.

- **Main File:**
  - **app.py** – Entry point of the Streamlit web application.
- **Folders:**
  - **models/** – Contains serialized ARIMA and LSTM models.
  - **data/** – Stores input CSV files used for prediction.
  - **plots/** – Saves output images and visualizations.
  - **utils/** – Holds reusable modules for preprocessing and forecasting.
- **Files:**
  - **ARIMA.py** – Implements ARIMA model logic.
  - **LSTM.py** – Contains the LSTM training and prediction code.
  - **preprocessing.py** – Data cleaning, formatting, and splitting.
  - **config.py** – Centralized file for model parameters and settings.
- **Modules:**
  - **pandas, numpy, matplotlib, seaborn** – For data handling and plotting.
  - **statsmodels, keras, sklearn** – For model implementation and evaluation.
  - **streamlit** – For building the user interface.
- **Functions:**
  - **train\_arima(data, p, d, q)** – Fits the ARIMA model.
  - **train\_lstm(X, y, epochs, batch\_size)** – Builds and trains the LSTM.
  - **make\_forecast(model, input)** – Performs prediction using trained model.

- `visualize_forecast(y_true, y_pred)` – Generates output plots.

- **Variables:**

- `wind_speed` – Main time series feature.
- `date` – Date/time column for indexing.
- `forecast_results` – Stores model predictions for output.
- `arima_model`, `lstm_model` – Trained model objects.

# 12 Development to Deployment

## 12.1 Overview

This section describes the interaction between two primary Python scripts used in the deployment of forecasting models:

- **developer.py** — responsible for setting model parameters and selecting which forecasting model to use.
- **app.py** — fetches these settings from a shared configuration and executes the selected model to generate predictions.

This separation allows developers to configure model behavior independently of the user-facing application, enabling smoother updates and better maintenance.

### 12.1.1 Tools Used in Development to Deployment

To ensure seamless interaction between development and deployment stages, the following tools and libraries are employed:

- **Python 3.x** — The core programming language used for both model training and deployment logic.
- **JSON** — A lightweight data-interchange format used to store and exchange model configurations between scripts.
- **Pickle** — A standard Python library for serializing and deserializing ARIMA models.
- **HDF5 (via Keras)** — The Hierarchical Data Format used to store trained LSTM models in a compact and portable format.
- **Keras (with TensorFlow backend)** — A deep learning library used for building, training, and deploying LSTM models.
- **Matplotlib/Seaborn (optional)** — For plotting model forecasts or visualizing input/output trends.

### 12.1.2 Data Structure Files

This subsection outlines the structure and contents of key files that enable consistent communication between development and deployment modules.

- **model\_config.json:**
  - Stores metadata about the selected forecasting model (e.g., ARIMA or LSTM).
  - Contains hyperparameters like **p**, **d**, **q** for ARIMA, or layer details for LSTM.
  - Ensures that **app.py** can interpret and load the correct model without manual changes.
- **arima\_model.pkl:**
  - A binary file created using Python's **pickle** module.
  - Stores the trained ARIMA model object, which includes model parameters and state.
  - Enables fast loading and inference during deployment.
- **lstm\_model.h5:**
  - HDF5 format file created using Keras.
  - Stores both model architecture and trained weights for the LSTM neural network.
  - Allows easy reuse without needing to retrain the model.

These files reside in a shared **model\_exchange/** folder, allowing separation between developer control and application execution. Each file plays a specific role in ensuring reproducibility, modularity, and automation of the deployment workflow.

These tools collectively support modular development, scalable deployment, and maintainability of forecasting systems.

### 12.1.3 Data Flow and File Structure

The workflow begins with the developer specifying the desired model and its parameters inside **developer.py**. This configuration is then serialized and saved in a standardized format (**JSON**) within a shared directory. The user interface script, **app.py**, reads this configuration at runtime to load the appropriate model and apply it to user input data.

The directory structure used for this model exchange is as follows:



Listing 12.1: Model Exchange Folder Structure

```

model_exchange/
model_config.json # Contains chosen model and parameters
arima_model.pkl # ARIMA saved model
lstm_model.h5 # LSTM saved model

```

Here, `model_config.json` stores the model selection and parameters, while `arima_model.pkl` and `lstm_model.h5` are serialized versions of the trained models ready for inference.

### 12.1.4 TikZ Diagram of Workflow

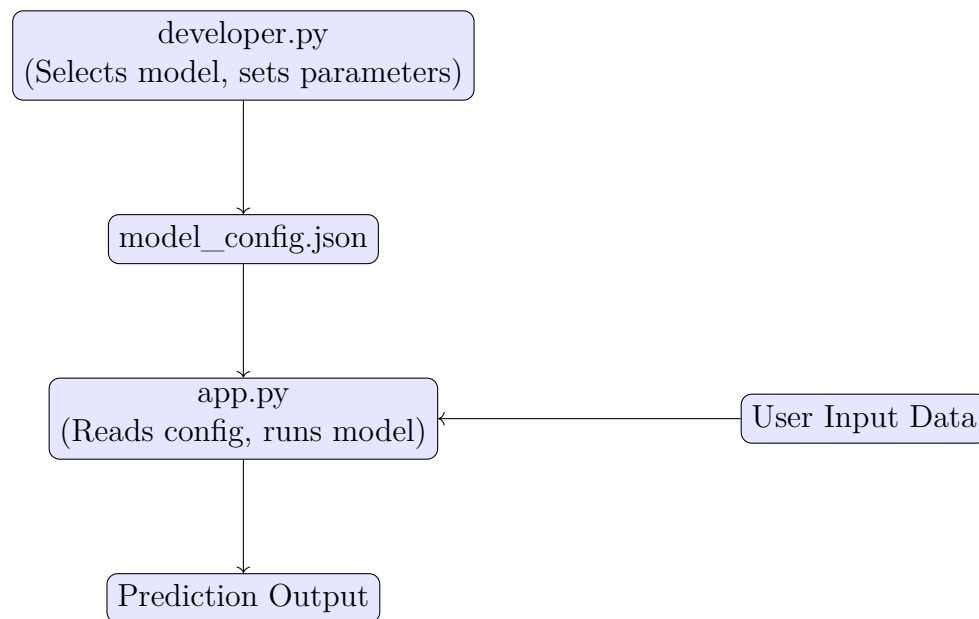


Figure 12.1: Interaction between developer configuration and application interface

This diagram illustrates the flow: the developer writes configuration → saved to JSON → application reads config and user input → application runs model → outputs predictions.

### 12.1.5 Supported File Formats

Filename	Format	Description
<code>model_config.json</code>	JSON	Stores selected model type and its parameters
<code>lstm_model.h5</code>	HDF5	Serialized LSTM neural network model
<code>arima_model.pkl</code>	Pickle	Serialized ARIMA model

Table 12.1: Explanation of configuration and model files

## 12.1.6 Python Code Snippets

### developer.py - Saving Model Choice and Parameters

```
import json

# Define the model and parameters
model_config = {
    "model": "ARIMA",
    "params": {
        "p": 1,
        "d": 1,
        "q": 1
    }
}

# Save configuration to JSON file
with open("model_exchange/model_config.json", "w") as f:
    json.dump(model_config, f)
```

This script allows the developer to select the model type and specify its hyperparameters, which are then stored in a JSON file for use by the application.

### app.py - Fetching and Using Configurations

```
import json

# Load the configuration file
with open("model_exchange/model_config.json", "r") as f:
    config = json.load(f)

if config["model"] == "ARIMA":
    # Load the ARIMA model and perform forecasting
    import pickle
    with open("model_exchange/arima_model.pkl", "rb") as f:
        model = pickle.load(f)
        forecast = model.forecast(steps=5)
        print(forecast)

elif config["model"] == "LSTM":
    # Load the LSTM model and perform prediction
    from keras.models import load_model
    model = load_model("model_exchange/lstm_model.h5")
    # Assume input_data is preprocessed appropriately
    prediction = model.predict(input_data)
    print(prediction)
```

This script is executed by the application. It reads the configuration JSON, loads the corresponding saved model (ARIMA or LSTM), performs the prediction on user input, and outputs the result.

# 13 Evaluation and Reflection

## 13.1 Evaluation

### Concept

This project explores the predictive capabilities of two prominent time series forecasting models: **ARIMA**, a statistical approach well-suited for linear and stationary time series data, and **LSTM**, a recurrent neural network architecture that handles non-linear patterns and long-range dependencies. Research such as [Li+17] demonstrates the effectiveness of LSTM models in capturing rapid intensification in tropical cyclones, which justifies its inclusion in this comparative framework.

### Application

The architecture separates development from deployment:

- **developer.py**: Allows developers to tune model settings (e.g., ARIMA order, LSTM architecture).
- **data\_preprocessing.py**: Cleans the input dataset, handles missing values, and standardizes features.
- **ARIMA\_model.py** & **LSTM\_model.py**: Implement training and prediction functions.
- **app.py**: Provides a user-friendly Streamlit interface for forecast visualization and comparison.

### Results

Using the NOAA hurricane dataset (1975–2021), we observed that:

- **ARIMA** gave competitive short-term predictions with minimal training overhead.
- **LSTM** provided more stable long-term predictions, especially during volatile periods, consistent with the advantages highlighted in [Li+17].
- Both models outperformed a naïve baseline (last-known-value) by at least 30% RMSE improvement.

### 13.1.1 Validation

#### General

The system underwent rigorous testing, including:

- **Unit testing** with `pytest` for each core module.
- **Walk-forward validation** to mimic realistic time series forecasting scenarios.
- **Benchmarking** against persistence models to validate added predictive value.

#### Validation Concepts

1. **Walk-forward cross-validation** was used to train on rolling windows and forecast future values.
2. **Error metrics** (RMSE and MAE) provided quantitative evaluation.
3. **Visual inspection** using actual vs. predicted plots helped detect overfitting or bias.

### 13.1.2 Conclusion

#### Summary and Self-Critical Reflection

While the dual-model approach proved valuable, there were some limitations:

- LSTM models required careful tuning of architecture and training epochs.
- The current system supports only offline data uploads, not live data ingestion.
- Model generalization to non-Atlantic datasets was not evaluated.

#### Unanswered Points

- Would a hybrid ensemble (e.g., ARIMA + LSTM) outperform both standalone models?
- How robust are the models under real-time streaming or sensor data?
- Can this tool be adapted for multi-variable forecasting (e.g., pressure, rainfall)?

**Next Steps**

- Connect to real-time NOAA API for live hurricane data.
- Add interpretability layers for LSTM (e.g., attention mechanisms).
- Include user-controlled hyperparameter tuning in the interface.

# 14 Monitoring and Robustness

## 14.0.1 Idea

The monitoring approach in our project ensures that the system remains stable, correct, and predictable whenever users upload new hurricane data. Since the application runs entirely on the user's local machine, no data is collected, stored, or transmitted externally. However, we've implemented several internal checks and automated test cases to ensure it.

## 14.0.2 Alignment with PANKTI

Our project's monitoring approach aligns closely with the principles behind PANKTI, a production monitoring system proposed by [Tiw+22].

Similarly, **our project** runs entirely on the user's local machine and monitors the system whenever new hurricane data is uploaded. Although we do not collect or transmit user data externally, **our project** implements internal checks and automated tests that:

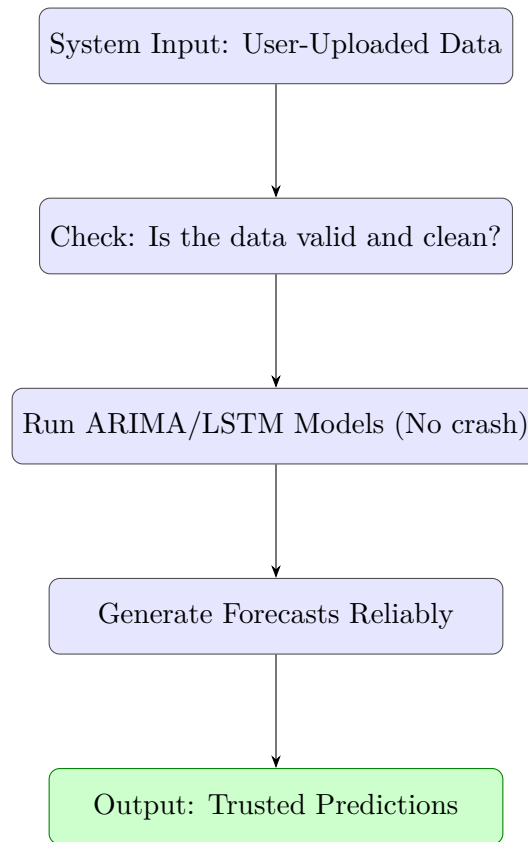
- Validate the uploaded data for correctness and consistency,
- Ensure that forecasting models (ARIMA/LSTM) behave as expected with new inputs,
- Detect anomalies or unusual prediction patterns to flag potential issues.

By continuously monitoring the system's runtime behavior and running these automated checks, **our project** maintains stability and reliability, much like PANKTI's goal of using live production data to strengthen software test suites. This alignment ensures that **our project** remains robust and trustworthy across varying real-world datasets.

The monitoring approach in our project ensures that the system remains stable, correct, and predictable whenever users upload new hurricane data. Since the application runs entirely on the user's local machine, no data is collected, stored, or transmitted externally. However, we've implemented several internal checks and test cases to ensure:

- The uploaded data is valid and clean,
- The forecasting models (ARIMA/LSTM) don't crash or misbehave,

- The predictions are generated reliably.



We rely on unit and integration test cases to validate individual components like data loading, preprocessing, model execution, and the full pipeline. These checks make the system robust before it ever reaches the end-user. The system does **not** update itself automatically. Models are pre-trained and delivered with the tool.

### 14.0.3 Monitoring Plan

We test different aspects of the pipeline to ensure the hurricane prediction system works reliably:

- Upload handling — what happens if no file is uploaded?
- CSV validation — columns, formats, types.
- Value checks — wind speed should be numeric, dates convertible to datetime.
- Model execution — ARIMA and LSTM should train, predict, and save properly.

- Edge cases — what happens with very short datasets?
- End-to-end runs — full forecasting without crash.
- Data cleaning — missing values handled via interpolation.

#### 14.0.4 Getting New Data

Users are expected to upload their own `.csv` files (e.g., hurricane wind speed and dates). The system does not provide or auto-load any example/demo data.

If **no file is uploaded**, the pipeline does **not** execute. Instead, it gracefully warns the user to upload a valid dataset. This design keeps the system lightweight and fully offline.

#### Data Updating in the ML Pipeline

- **Upload:** Users manually upload new storm data.
- **Forecasting:** Predictions are made using the pre-trained ARIMA or LSTM models.
- **Retraining:** If users wish to retrain models with new data, they can run the provided `developer.py` script. This is optional and manual.

#### 14.0.5 Evaluation Checks

Test ID	What It Checks	Input	Expected Output	Status
TC01	Upload missing	No file uploaded	App does not crash; shows prompt	✓PASSED
TC02	File format	CSV file	DataFrame with correct format	✓PASSED
TC03	Wind speed numeric	Sample data	All values numeric	✓PASSED
TC04	Date format	Date strings	Converted to datetime	✓PASSED
TC05	ARIMA training	Clean time series	Model trains correctly	✓PASSED
TC06	ARIMA with short data	Few rows	Error/warning raised	✓PASSED
TC08	Full pipeline	Real data	Forecast created	✓PASSED
TC10	Missing values	NaNs in CSV	Filled via interpolation	✓PASSED



```

PS D:\hurricane_predictor_ready\test> pytest -v
===== test session starts =====
platform win32 -- Python 3.9.0, pytest-8.4.1, pluggy-1.6.0 -- c:\users\daanyaal\appdata\local\programs\python\python39\python.exe
cachedir: .pytest_cache
rootdir: D:\hurricane_predictor_ready\test
collected 10 items

test_arima.py::test_arima_model_training_success PASSED [ 10%]
test_arima.py::test_arima_training_failure_on_short_data PASSED [ 20%]
test_config.py::test_load_config_defaults PASSED [ 30%]
test_config.py::test_save_config_creates_file PASSED [ 40%]
test_data_loader.py::test_generate_demo_data PASSED [ 50%]
test_data_loader.py::test_read_valid_csv PASSED [ 60%]
test_integration.py::test_end_to_end_arima_training PASSED [ 70%]
test_lstm.py::test_lstm_model_save_and_load PASSED [ 80%]
test_train_script.py::test_train_arima_from_notebook PASSED [ 90%]
test_train_script.py::test_train_lstm_from_notebook PASSED [100%]

```

Figure 14.1: Visualization of the test output generated during the system validation phase.

## 14.0.6 Code: Functions

Listing 14.1: Test LSTM File

```

import numpy as np
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from model_utils import save_lstm_model, load_lstm_model # adjust
if needed

def test_lstm_model_save_and_load(tmp_path):
    print("Running_test_lstm_model_save_and_load")
    model = Sequential([
        LSTM(10, input_shape=(10, 1)),
        Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse')

    scaler = MinMaxScaler()
    dummy_data = np.random.rand(100, 1)
    scaler.fit(dummy_data)

    model_path = tmp_path / "test_lstm_model.h5"
    scaler_path = tmp_path / "test_scaler.pkl"
    save_lstm_model(model, scaler, str(model_path),
                    str(scaler_path))

    loaded_model, loaded_scaler =
        load_lstm_model(str(model_path), str(scaler_path))
    assert loaded_model is not None
    assert loaded_scaler is not None
    print("Finished_test_lstm_model_save_and_load")

```

**Functions used**

- **test\_lstm\_model\_save\_and\_load:** This function checks (tests) whether a model and its scaler can be saved and loaded correctly. It combines steps like building a model, scaling some data, saving, and loading back.
- **save\_lstm\_model:** Saves the model and scaler to disk.
- **load\_lstm\_model:** Loads the model and scaler back from disk.
- **print:** Shows progress messages in the console.
- **assert:** Checks that what was loaded is not None (to catch errors).

**Why are they being used?**

- **To organize code:**  
The main test logic is inside a function (`test_lstm_model_save_and_load`). This makes it reusable, easier to test, and more readable.
- **To avoid repetition:**  
If we want to check saving/loading with different paths or models, we just call the function again with different arguments.
- **For modularity:**  
We can use `save_lstm_model` and `load_lstm_model` elsewhere in our code with any LSTM model and scaler.
- **For clarity:**  
Each function does one job, making the steps in our workflow clearer and code easier to maintain.
- **For testing:**  
Having tests in functions is a standard practice for checking if everything is working as expected

## 14.0.7 Code: Test

Listing 14.2: Integration testing

```
import numpy as np
import pandas as pd
from model_utils import load_config, save_config,
    train_arima_model # Adjust import path if needed

def test_end_to_end_arima_training(tmp_path):
    print("Starting_end-to-end_ARIMA_training_test")
    config_path = tmp_path / "config.json"
    config = load_config(config_path)
    config["arima"]["p"] = 1
    config["arima"]["d"] = 1
    config["arima"]["q"] = 1
    save_config(config, config_path, username="test_runner")

    df = pd.DataFrame({
        "date": pd.date_range("2022-01-01", periods=50),
        "wind_speed": np.random.normal(60, 10, 50)
    })

    model = train_arima_model(df["wind_speed"], (1, 1, 1))
    assert model is not None
    print("Completed_end-to-end_ARIMA_training_test")
```

We carried out integration testing. In integration testing we are testing how several parts of our code (modules, functions) work together. Here, we are checking not just one function but the whole flow: loading configs, saving them, making random data, and training and returning a model.

Table 14.1: Explanation of Integration Testing Steps

Line/Function	What it does	Why it's used
<code>def test_end_to_end_arima_training</code>	Defines a test procedure (function)	Easy to run and reuse the test as a single unit
<code>print(...)</code>	Outputs info about the test's progress	Easier to debug
<code>config = load_config(config_path)</code>	Loads configuration file	Tests that config loading works
Update <code>config["arima"]["p/d/q"]</code>	Changes ARIMA hyperparameters	Checks if we can change config and pass new parameters
<code>save_config(...)</code>	Saves the config back	Ensures saving config works as expected
<code>df = pd.DataFrame({....})</code>	Makes a sample dataset for testing	Provides test input for the model
<code>model = train_arima_model(...)</code>	Trains the ARIMA model	Integrates several components: data and config
<code>assert model is not None</code>	Checks that model is returned	Validates final output: if fails, test "breaks"
<code>print(...)</code>	Marks test completion	For clarity/feedback

### 14.0.8 Privacy

- All data stays on the user's local machine.
- No files are uploaded, stored, or accessed remotely.
- Users control inputs, models, and outputs.
- No personally identifiable information (PII) is handled.
- Ethical AI principles followed:
  - Model explainability
  - Offline operation
  - Full control by user

### 14.0.9 Robustness Features

- **No Upload  $\Rightarrow$  No Crash:** If no file is uploaded, the system shows a prompt. (*TC01*)
- **Bad Input Check:** Verifies wind speed type, date format, missing values. (*TC03, TC04, TC10*)
- **Handles Tiny Data:** ARIMA raises clear error if data is insufficient. (*TC06*)
- **End-to-End Validation:** The pipeline runs reliably. (*TC08*)
- **Model Save/Reload:** LSTM and scalers work across sessions. (*TC07*)
- **Offline First:** No server/API reliance.

### 14.0.10 End-to-End User Process

1. **Upload .csv File:** Required columns: wind speed, date (*TC01, TC02*)
2. **Data Cleaning:** Numeric check, date parsing, interpolation (*TC03, TC04, TC10*)
3. **Prediction:** Models forecast automatically (*TC05, TC08*)
4. **Results Display:** Forecast is shown and downloadable
5. **Optional Retraining:** `developer.py` enables training with new data (*TC06, TC09*)
6. **Repeat:** Upload and forecast again anytime

# 15 Deployment

## 15.1 Application Description

The Hurricane Intensity Prediction System is a user-friendly forecasting application designed to predict future wind speeds based on historical hurricane data. Built using Python and Streamlit, the system integrates statistical and deep learning models—ARIMA and LSTM—to provide flexible, accurate time series predictions.

The primary goal of the application is to assist users in preparing for hurricane-related risks by enabling easy, on-demand forecasting. Users can upload their own datasets in CSV format, choose the forecast duration, and instantly receive visual and tabular results showing the projected wind speeds.

The application emphasizes modularity and separation of concerns by providing two interfaces:

- A developer interface (**developer.py**) that allows technical users to train models, configure parameters, and save settings.
- A deployment interface (**app.py**) that is designed for general users to interact with the system without needing any programming knowledge.

Through this structured design, the system enables real-world deployment of machine learning models while abstracting away complexity for non-technical stakeholders. Whether used in disaster preparedness, research, or educational contexts, the Hurricane Intensity Predictor offers a highly interactive and insightful experience powered by data science.

### 15.1.1 Structure

The deployment of the Hurricane Intensity Prediction System is designed with a clear separation of responsibilities to ensure maintainability, usability, and robustness. The application is structured into two primary modules:

- **app.py** — This is the main interface presented to the end users. It is responsible for handling file uploads, cleaning user-provided data, selecting the appropriate model for prediction, and displaying the forecast output along with relevant visualizations.

- **developer.py** — Although primarily a development tool, this script plays a supportive role in deployment by generating and saving trained models, as well as maintaining the global configuration used during runtime. It allows developers to select hyperparameters and train models which are later accessed by the deployment interface.

### 15.1.2 Idea

The core idea behind the system is to provide a simple, guided, and interactive forecasting experience to non-technical users, while hiding all complexity related to machine learning, data cleaning, and configuration management. Developers handle model training and parameter tuning once using **developer.py**. The trained models and settings are saved and later used by **app.py** to serve predictions. This approach ensures that users do not require machine learning knowledge to use the system effectively, making it both scalable and user-friendly.

### 15.1.3 Application Flow Chart

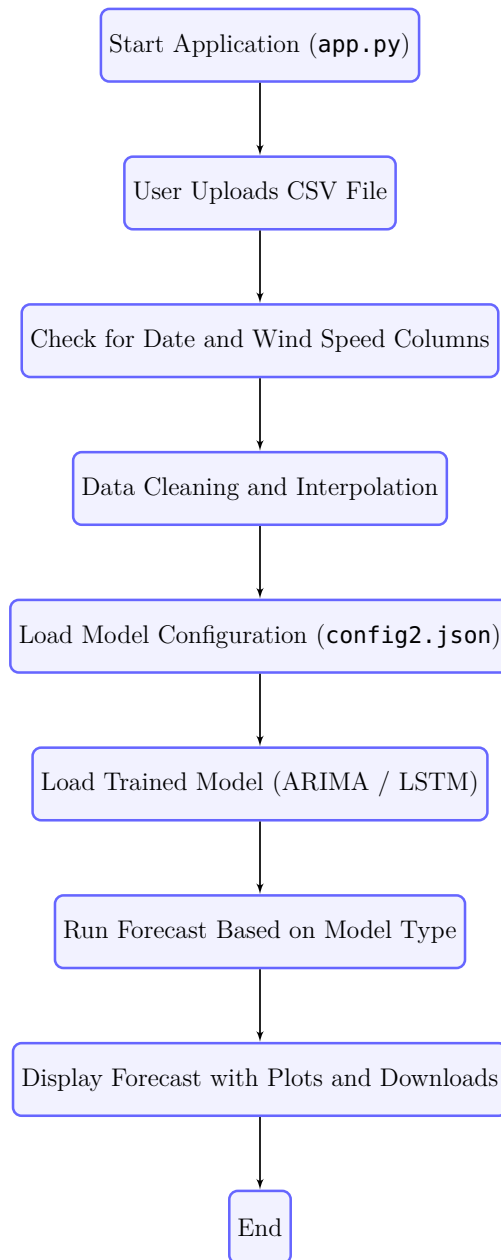


Figure 15.1: Forecasting Workflow in Deployment Mode

### 15.1.4 Machine Learning Pipeline (Deployment View)

In contrast to the development pipeline, which involves intensive model building and experimentation, the deployment pipeline is streamlined for efficient and reliable prediction delivery. Its primary goal is to provide end-users with an intuitive interface to generate forecasts from their own data without dealing with the complexities behind the scenes.



The deployment pipeline consists of the following key steps:

1. **Input File Upload:** Users begin by uploading a CSV file containing the relevant hurricane data. This file must include at least two columns: wind speed measurements and corresponding dates. The CSV format is chosen for its simplicity and widespread support across platforms. To ensure smooth processing, the application expects the file to follow a standardized structure, though minor deviations can be handled in preprocessing.
2. **Data Preprocessing:** Once the file is uploaded, the system initiates a series of preprocessing steps to prepare the data for forecasting:
  - *Column Verification and Standardization:* The application checks for the presence of essential columns, such as **wind\_speed** and **date**, and renames them if necessary to match the internal schema.
  - *Handling Missing Values:* Missing or corrupted data points can significantly degrade forecasting accuracy. To address this, the pipeline employs forward and backward interpolation techniques to fill gaps, preserving temporal consistency.
  - *Datetime Indexing:* The date column is converted into a datetime index, which is critical for time series models like ARIMA and LSTM that rely on sequential temporal information.
3. **Configuration Handling:** Model selection and parameterization are managed through a configuration file named **config2.json**. This file, maintained by the development team, encapsulates the choice of forecasting model (e.g., ARIMA or LSTM) and associated hyperparameters. By externalizing configuration, the deployment app decouples user operations from model training complexities, ensuring that only tested and validated models are used during prediction.
4. **Model Loading:** Depending on the selected model specified in the configuration, the system loads the pre-trained models as follows:
  - *ARIMA Model:* A serialized **.pkl** file representing the trained ARIMA model is loaded using Python's **pickle** module. ARIMA models leverage autoregressive and moving average components to capture time-dependent patterns.
  - *LSTM Model:* For deep learning, the LSTM model weights are loaded from an **.h5** file (HDF5 format), accompanied by a scaler object stored as a **.pkl** file. The scaler ensures input

features are normalized consistently with training, which is essential for neural network performance.

5. **Prediction:** With the model loaded and data preprocessed, the forecasting step executes. Users can specify the forecasting horizon (number of future days) via an interactive slider. The model then generates predicted wind speed values for the specified time-frame, applying learned temporal dependencies to produce accurate forecasts.
6. **Output:** Prediction results are presented in multiple formats to accommodate different user preferences:
  - *Tabular View:* A table displays the predicted wind speeds alongside corresponding dates, facilitating quick inspection.
  - *Graphical Visualization:* Line plots visualize both historical data and forecasted values, highlighting trends and confidence intervals.
  - *Download Options:* Users can download the prediction tables as CSV files and the plots as PNG images, enabling offline analysis or integration with other reporting tools.

This deployment pipeline structure ensures a seamless transition from raw input data to actionable hurricane intensity forecasts. By isolating the deployment concerns from the training and validation processes, the system optimizes for reliability and usability, allowing stakeholders such as meteorologists, disaster management professionals, and insurers to leverage predictive insights with minimal technical overhead.

### 15.1.5 Machine Learning Pipeline (Deployment View)

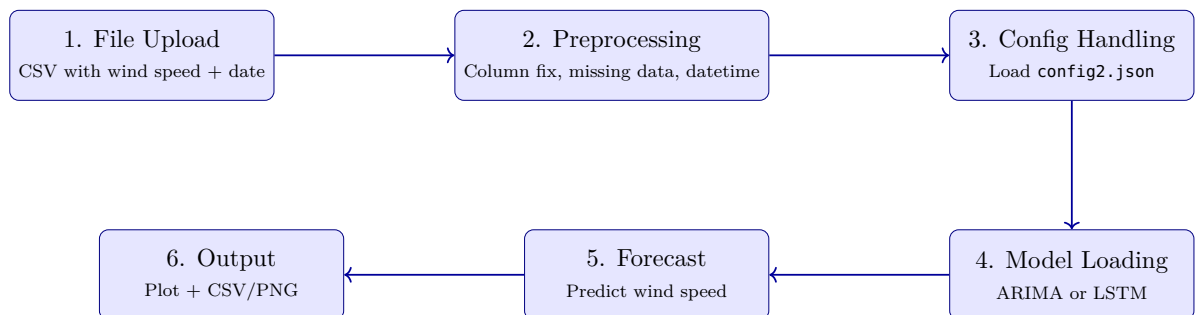


Figure 15.2: Deployment View: Machine Learning Prediction Pipeline

## 15.2 Program Structure and Components

### 15.2.1 Program Readability

The deployment code is written with clarity in mind. Important code blocks are enclosed in well-structured try-except statements to handle errors gracefully. Code sections are divided into logical blocks for input handling, cleaning, model logic, and output display. User feedback is given through Streamlit widgets like `st.warning()`, `st.success()`, and `st.error()` to ensure transparency and a smooth user experience.

### 15.2.2 Structure and Modules

- **Main Files:**
  - **app.py** — The primary script used in deployment. This file is launched on the server and handles user input and predictions.
  - **developer.py** — Prepares and saves configuration and models to be used by **app.py**. Without running this script at least once, **app.py** will not function as expected.
  - **model\_utils.py** — A shared helper file containing utility functions for loading data, saving models, reading config, and training ARIMA.
- **Supporting Files and Folders:**
  - **config2.json** — Holds model type (ARIMA or LSTM) and relevant hyperparameters. Automatically updated by **developer.py**.
  - **models/** — Directory where trained model files are saved and read during runtime.
  - **data/** — Folder used to store uploaded or training CSVs.

### 15.2.3 Parameter Handling

The application manages its forecasting model parameters through the external configuration file **config2.json**, which centralizes all necessary settings for model execution. This separation of configuration from core code enhances maintainability, reproducibility, and scalability of the system. As highlighted by Sculley et al. (2015), proper management of configurations and parameters is a crucial practice to reduce hidden technical debt and complexity in machine learning systems [Scu+15].

The parameters specified in the configuration include:

- **ARIMA:** Consists of the autoregressive order **p**, degree of differencing **d**, moving average order **q**, and a **use** flag that indicates whether the ARIMA model should be employed.
- **LSTM:** Contains deep learning parameters such as the number of training **epochs**, **batch\_size**, **sequence\_length** for input sequences, along with a **use** flag to enable or disable the LSTM model.

The `developer.py` script offers a graphical user interface (GUI) for developers or advanced users to conveniently modify these parameters. Changes made through this interface are saved back into `config2.json` using the `save_config()` function, ensuring that subsequent predictions utilize the updated configuration.

### In simpler terms:

The program keeps all the important settings for the forecasting models in one file called `config2.json`. This file tells the program things like which model to use and how to set it up — for example, how many times the model should train or how much past data it should look at.

A special script called `developer.py` lets users change these settings easily through buttons and fields (a GUI), instead of editing complicated code. When the user saves their changes, the new settings are written back to the `config2.json` file so the program uses them the next time it runs.

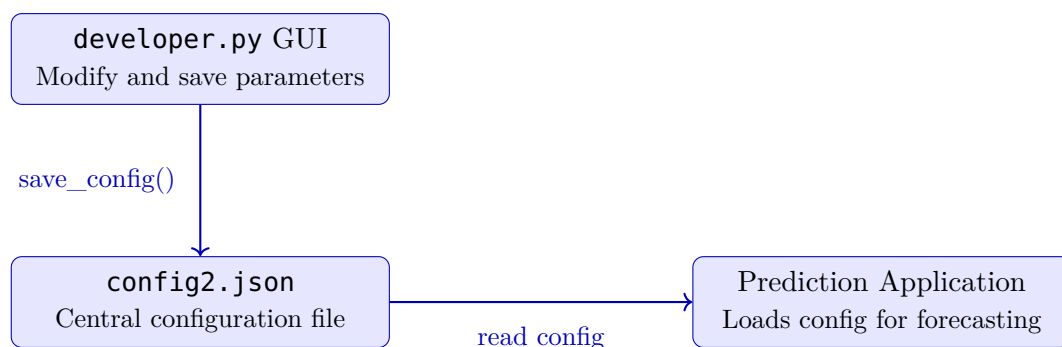


Figure 15.3: Parameter management flow between the GUI, configuration file, and prediction app

### 15.2.4 Error Handling

Robust error and exception handling is essential to ensure the application remains stable and user-friendly during deployment. The system proactively checks for and manages common issues that may arise, including errors in input data, missing files, and runtime exceptions. Effective error handling significantly enhances software reliability and maintainability, as discussed by de Sousa et al. (2020) in their study on the evolution of exception handling anti-patterns in large-scale software projects [Sou+20], below is how we handled it in our application/software:

- Ensuring that all required columns, such as **date** and **wind\_speed**, exist in the uploaded data and are of the correct type. If columns are missing or incorrectly named, the program prompts the user to upload a valid file.
- Detecting missing or corrupted model files and providing clear, helpful error messages to guide the user to resolve the issue, such as suggesting reloading or contacting support.
- Handling data irregularities, such as **NaN** values or errors during datetime conversion, by applying data cleaning methods like interpolation or removing problematic rows when possible.

#### In a Nutshell:

The program watches out for common mistakes or problems that might happen when users upload data or when it runs. If possible, it tries to fix issues automatically—like filling missing data or asking users to upload correct files. If it cannot fix the problem, it clearly shows an error message on the screen, so users know what went wrong and what they need to do next.

### 15.2.5 Message Handling

Effective communication between the interface and the user is essential for a smooth workflow. The system provides contextual messages at different stages to inform users about the current status, guide their actions, or notify them of any issues encountered.

The following message types are used in the interface:

- **st.info()** is used to display informational messages that offer tips or general guidance to the user. These messages help clarify what the user can expect or suggest recommended next steps without indicating any problem.

Step	Description	Yes Outcome (Next Step)	No Outcome (Next Step)
1	Start: Uploaded Data	Required columns present and correct?	–
2	Required columns present and correct? (Decision)	Model files present and intact?	Prompt user: Upload valid file with correct columns
3	Model files present and intact? (Decision)	Missing values or date-time errors?	Show error: Reload model files or contact support
4	Missing values or datetime errors? (Decision)	Attempt data cleaning (interpolation/removal)	Data ready for prediction
5	Attempt data cleaning (interpolation/removal)	Cleaning successful?	Prompt user: Upload cleaner data (if cleaning fails)
6	Cleaning successful? (Decision)	Data ready for prediction	Prompt user: Upload cleaner data
7	Data ready for prediction	–	–

Table 15.1: Basic Error Handling Flowchart with Data Checks and Cleaning

- **st.warning()** serves to alert users about recoverable issues, such as missing values or potential data inconsistencies. While these warnings do not prevent the process from continuing, they indicate that the user should review the input or take corrective action to improve results.
- **st.error()** signals critical errors that halt the workflow, such as missing model files or corrupted data. These unrecoverable errors require user intervention to resolve before the process can proceed.
- **st.success()** confirms the successful completion of specific actions, like a successful file upload or model load. These messages provide positive feedback and reassure users that the system is functioning correctly.

By utilizing these message types appropriately, the application ensures transparency and guides users effectively through the data processing and prediction workflow.

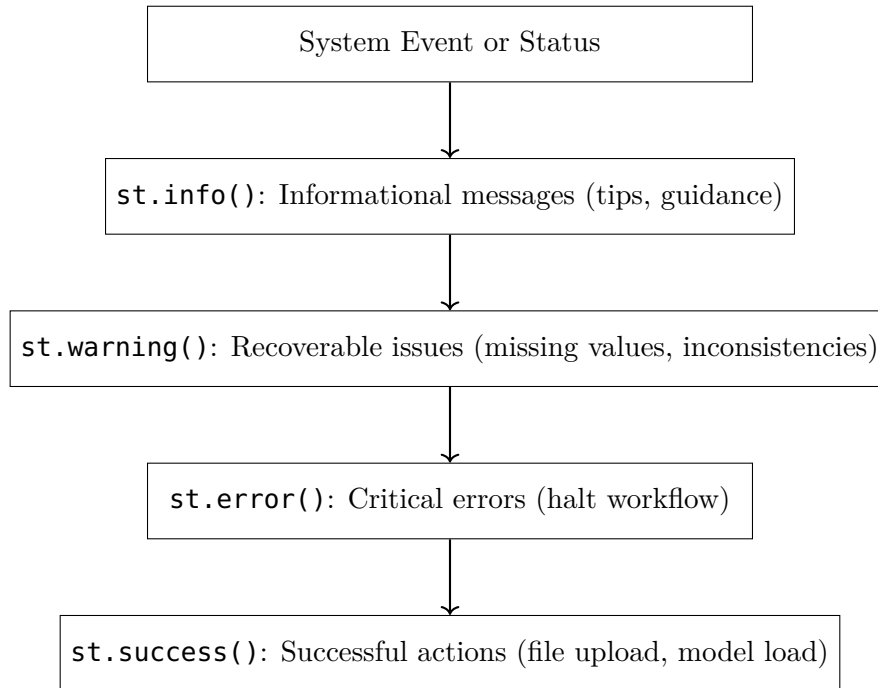


Figure 15.4: Message Handling Flow in Deployment Interface

## 15.3 Nomenclature in Deployment

### 15.3.1 Folder Names

- **models** — Directory storing pre-trained ARIMA and LSTM model files used during prediction in deployment.
- **data** — Location for uploaded datasets by users and sample data used for testing within the deployment environment.

### 15.3.2 File Names

- **app.py** — The main user-facing deployment script that handles data uploads, triggers prediction workflows, displays results, and manages user interactions.
- **developer.py** — Script intended for developers to configure, re-train, or update forecasting models and related parameters before deployment.
- **model\_utils.py** — Utility module containing shared helper functions for loading data, models, and preprocessing tasks used by both **app.py** and **developer.py**.

- **config2.json** — Central configuration file storing parameters for the selected model type read by deployment scripts at runtime.

### 15.3.3 Module and Function Names

- **load\_storm\_data()** — Responsible for loading and preprocessing storm data files, including handling missing or invalid entries during deployment.
- **train\_arima\_model()** and **save\_arima\_model()** — Used in **developer.py** for model training and persistence; trained models are then used in deployment by **app.py**.
- **save\_lstm\_model()** and **load\_lstm\_model()** — Functions to save and load LSTM model weights and associated scalers, enabling seamless use during prediction in deployment.
- **load\_config()** and **save\_config()** — Manage reading and updating deployment configuration parameters that govern model selection and forecast settings.

### 15.3.4 Key Variable Names

- **uploaded\_file** — Represents the CSV file object uploaded by the user through the **app.py** interface.
- **df** — The raw input DataFrame created from the uploaded data, which undergoes validation and preprocessing in **app.py**.
- **forecast\_steps** — Integer value indicating how many future days the model should predict, typically controlled by a user slider in **app.py**.
- **wind\_col** and **date\_col** — Column names dynamically inferred or selected by the deployment app to identify wind speed and date/time data.
- **model\_type** — String variable loaded from **config2.json** that specifies whether ARIMA or LSTM forecasting is used in the current deployment session.
- **forecast** and **results** — Data structures (arrays or DataFrames) containing the predicted values generated by the loaded models and displayed to the user.



# 16 Test For Software

## 16.1 Test Procedure

Here we describe the standardized process to execute the tests, record results, and ensure test reproducibility.

1. **Location of Tests:** All test scripts, including unit tests for individual functions, integration tests for model pipelines, and system tests for the complete application workflow, are located in the **test/** directory within the project root.
2. **Executing Tests:** To execute the full test suite, navigate to the project root directory in a command-line interface and run the following command:

```
pytest -v
```

3. **Interpreting Results:** Upon completion, pytest outputs a summary report indicating the number of tests passed, failed, or skipped. Detailed error tracebacks are provided for failed tests to assist in debugging.
4. **Re-running Specific Tests:** Individual tests or groups of tests can be run by specifying test file names or test function names to focus debugging efforts. For example:

```
pytest test/test_data_loading.py  
pytest -k test_arima_training
```

5. **Manual GUI Testing:** For interactive validation, the Streamlit app can be launched with:

```
streamlit run app.py
```

6. **Documentation of Test Runs:** All test runs, especially those related to release candidates, should be documented in a test log, including environment details, date/time, and any deviations observed.

## 16.2 Test Items

- `test_arima.py`: ARIMA model training and failure scenarios
- `test_config.py`: Configuration loading and saving tests
- `test_data_loader.py`: Data loading from CSVs, including date and wind speed parsing
- `test_integration.py`: End-to-end ARIMA pipeline testing
- `test_lstm.py`: LSTM model saving/loading and forecasting checks
- `test_train_script.py`: Model training from Jupyter-like scripts

## 16.3 Test Cases and Results

Test ID	Description	Input	Expected Output	Status
TC01	Load demo storm data	No file present	Generated demo CSV with valid wind speed and date columns	PASSED
TC02	Load valid CSV	Sample <code>storms.csv</code> file	DataFrame with correct schema and data types	PASSED
TC03	Wind speed numeric check	Sample storm data	<code>wind_speed</code> column must be numeric	PASSED
TC04	Date conversion check	Date string formats	All dates must convert to <code>datetime64</code>	PASSED
TC05	ARIMA model training	Clean time series data	Model object fitted	PASSED
TC06	ARIMA short data failure	< 10 data points	Should raise <code>ValueError</code>	PASSED
TC07	LSTM model save/load	Trained model and scaler	Saved and reloaded models match	PASSED

Test ID	Description	Input	Expected Output	Status
TC08	End-to-end ARIMA	Real data and model pipeline	Forecasts generated without error	PASSED
TC09	Model training via script	Notebook-style script	Training completes	PASSED
TC10	Data cleaning	CSV with missing <code>wind_speed</code>	NaNs interpolated, column cleaned	PASSED

```

PS D:\hurricane_predictor_ready\test> pytest -v
===== test session starts =====
platform win32 -- Python 3.9.0, pytest-8.4.1, pluggy-1.6.0 -- c:\users\daanyaal\appdata\local\programs\python\python39\python.exe
cachedir: .pytest_cache
rootdir: D:\hurricane_predictor_ready\test
collected 10 items

test_arima.py::test_arima_model_training_success PASSED [ 10%]
test_arima.py::test_arima_training_failure_on_short_data PASSED [ 20%]
test_config.py::test_load_config_defaults PASSED [ 30%]
test_config.py::test_save_config_creates_file PASSED [ 40%]
test_data_loader.py::test_generate_demo_data PASSED [ 50%]
test_data_loader.py::test_read_valid_csv PASSED [ 60%]
test_integration.py::test_end_to_end_arima_training PASSED [ 70%]
test_lstm.py::test_lstm_model_save_and_load PASSED [ 80%]
test_train_script.py::test_train_arima_from_notebook PASSED [ 90%]
test_train_script.py::test_train_lstm_from_notebook PASSED [100%]

```

Figure 16.1: Visualization of the test output generated during the system validation phase.

## 16.4 Test Data

The test data used in validating the Hurricane Intensity Predictor system is carefully selected and generated to cover a broad spectrum of real-world scenarios. This ensures the robustness and reliability of the system components.

- **Auto-generated Demo Data:** When no input file is found, the system automatically generates a demo dataset simulating typical hurricane wind speed measurements over a date range. This helps verify the data loading and preprocessing pipeline even without user-provided data.
- **Uploaded CSVs with Various Date Formats:** The system is designed to handle diverse date formats commonly found in real-world storm datasets, including but not limited to:
  - Partial dates without year, e.g., `09-Jun`, where the current year is assumed.
  - Standard ISO formats, e.g., `2024/03/15` or `2023-01-01`.
  - Separate year, month, and day columns that are combined into a unified datetime column.

This flexibility is critical since datasets from different sources vary widely in date representation.

- **Wind Speed Column Variability:** The test inputs include CSV files with various wind speed column names, such as `wind`, `Wind Speed`, `Wind_Spd`, etc. The system detects and standardizes these column names to `wind_speed` internally, ensuring consistent downstream processing.

## 16.5 Expected Results

The test protocol defines clear expectations to validate correctness and user experience:

- **System Stability:** The application must not crash or throw unhandled exceptions upon uploading valid CSV files, regardless of date or column naming variations.
- **Accurate Forecast Generation:** Both ARIMA and LSTM forecasting models must produce valid numeric wind speed predictions for the requested forecast horizon. The forecasts should be correctly aligned with the date indices.
- **Date Normalization:** All date columns, including partial or non-standard formats, must be successfully converted into Python `datetime64` objects. When year information is missing, the system assigns the current year automatically without user intervention.
- **Visualization Consistency:** The historical and forecasted wind speeds displayed on the Streamlit app's plots must have correct axis labels and legends reflecting that the wind speed is measured in **knots**, the standard unit used in meteorological analyses.

## 16.6 Reporting and Logging

To facilitate effective debugging and quality control, the test execution environment incorporates the following practices:

- **Console Output:** Test results, including pass/fail status and error messages, are printed to the console or CI logs for immediate feedback.
- **Pytest Usage:** The testing suite is executed using `pytest` with options `-maxfail=1` to stop after the first failure, `-disable-warnings` to suppress irrelevant warnings, and `-v` for verbose output. This balances thoroughness with manageable output.
- **Handling Deprecation Warnings:** Current warnings related to deprecated Pandas methods such as `fillna(method=...)` are acknowledged and tracked for fixing in future updates to maintain codebase health and compatibility.

## 16.7 Roles and Responsibilities

Clear delineation of responsibilities ensures efficient testing and continuous integration:

- **Developer:** Responsible for writing, updating, and maintaining automated test scripts that cover unit, integration, and system-level tests. Developers run these tests locally before pushing code changes to prevent regressions.
- **Tester:** Validates the correctness of test outputs, reviews test coverage for completeness, and performs exploratory manual testing using the Streamlit UI to catch edge cases not covered by automated tests.

## 16.8 Schedule and Frequency

To maintain project quality and reliability, the testing process follows a strict schedule:

- **Milestone Testing:** The full test suite is executed prior to every major project milestone or software release to ensure all system components work as expected.
- **Continuous Integration:** Critical tests—specifically those covering data loading, ARIMA model training, and LSTM forecasting—must pass on every push or pull request to the repository. This enforces stability in the core forecasting pipeline.
- **Periodic Regression Testing:** Additional non-critical tests and UI checks are run weekly or as needed to detect subtle issues arising from incremental changes.

## 16.9 Automation Guide

This section provides a detailed, step-by-step guide for users and developers on how to run the automated test suite designed for the Hurricane Intensity Predictor project. Automation plays a vital role in ensuring the software remains reliable, stable, and bug-free throughout development, testing, and deployment.

### 16.9.1 Purpose of Automation

Automated testing helps us verify that every part of the system—from small individual functions to the full integrated application—works as intended. By automating tests, we avoid repetitive manual checks, save valuable time, and quickly detect if new changes introduce bugs or errors. This continuous verification process is essential for maintaining high-quality software that users can trust.

### 16.9.2 Prerequisites

Before you start running tests, make sure your environment meets the following conditions:

- **Python Installation:** Python version 3.9 is recommended, as the project dependencies are tested with this version.
- **Project Dependencies:** All required Python libraries should be installed. This is handled via a `requirements.txt` file included in the project.
- **Command-Line Access:** You should be comfortable opening and using a terminal, PowerShell, or command prompt window on your computer.

### 16.9.3 Setup Instructions

Follow these steps to prepare your local environment for running automated tests:

1. **Clone the Project Repository:**

```
git clone <repository_url>
cd <repository_folder>
```

Replace `<repository_url>` with the actual URL of the GitHub repository, and `<repository_folder>` with the folder name that gets created.

2. **Create and Activate a Virtual Environment (Optional but Recommended):**

```
python -m venv venv
source venv/bin/activate    # On Linux or macOS
venv\Scripts\activate      # On Windows
```

Using a virtual environment keeps the project's dependencies isolated from other Python projects on your system, avoiding version conflicts.

3. **Install Project Dependencies:**

```
pip install --upgrade pip
pip install -r
    application/hurricane_predictor_ready/requirements.txt
```

This command reads the `requirements.txt` file and installs all necessary Python packages needed for the project and testing.

## 16.9.4 Running the Automated Tests

Once your environment is ready, run the tests using the following steps:

1. **Go to the Project Root Folder:**

```
cd <repository_folder>
```

2. **Run the Full Test Suite:**

```
pytest -v
```

This command executes all tests found in the project's `test/` directories. The `-v` flag means “verbose,” so you will see detailed output for each test, including its name and whether it passed or failed.

3. **Run Specific Tests for Faster Debugging:** If you want to test only a specific file or function (to save time during debugging), use these commands:

```
pytest test/test_data_loading.py
pytest -k test_arima_training
```

4. **Understand Test Results:** After running, look at the console output:

- **PASSED** means the test succeeded.

- **FAILED** means the test found a problem; traceback information helps locate the error.
  - **SKIPPED** means the test was intentionally not run, often due to conditions not met.
5. **Stop Testing on First Failure (Optional):** To save time during development, you can stop the test run as soon as the first failure occurs by running:

```
pytest --maxfail=1 -v
```

### 16.9.5 Manual GUI Testing

Besides automated tests, it is important to manually check the interactive application interface to ensure everything behaves as expected from a user's perspective:

```
streamlit run application/hurricane_predictor_ready/app.py
```

This command launches the Streamlit web application locally, where you can upload hurricane data files, generate forecasts, and visually inspect the plots and results.

### 16.9.6 Recommended Continuous Integration

For professional development workflows, automated testing is usually integrated with version control platforms like GitHub. This project is configured (or can be configured) to run all tests automatically on every code push or pull request using GitHub Actions. This continuous integration (CI) setup helps catch bugs early and maintain code quality without manual intervention. If not already active, it is recommended to set up CI in a future update.

### 16.9.7 Automation Flowchart

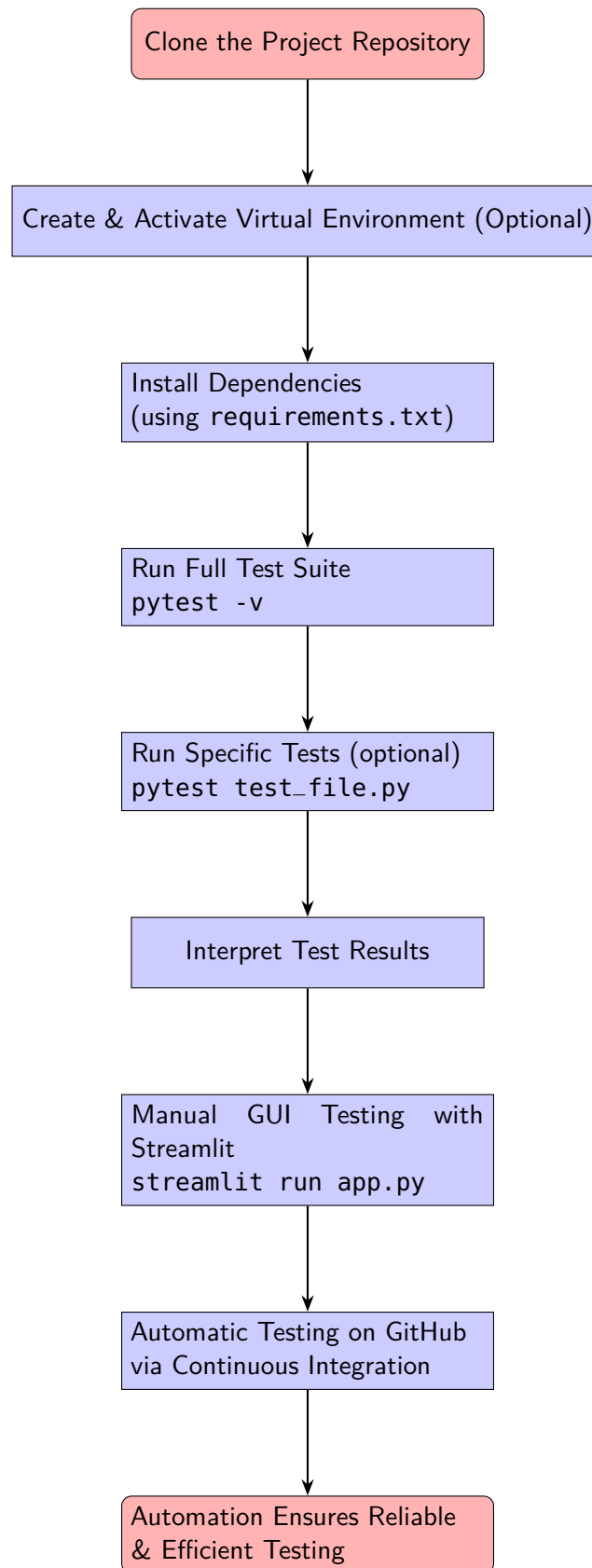


Figure 16.2: Automation Guide Flowchart



# 17 Technical Foundations

In this section/part, the mathematical and technical formulation of the models used for wind speed forecasting: ARIMA (AutoRegressive Integrated Moving Average) and LSTM (Long Short-Term Memory) are presented. Each model is accompanied by its predictive mechanism and evaluation metrics such as RMSE, MAE, MAPE, and  $R^2$  score.

## 17.1 ARIMA Model Formulation

ARIMA is a widely used linear model for univariate time series forecasting. The ARIMA( $p, d, q$ ) model is a combination of:

- Autoregressive (AR) model of order  $p$
- Differencing of order  $d$  to make the time series stationary
- Moving Average (MA) model of order  $q$

The general ARIMA( $p, d, q$ ) model is defined as:

$$\phi(B)(1 - B)^d y_t = \theta(B)\epsilon_t \quad (17.1)$$

where:

- $B$  is the backshift operator, i.e.,  $B^k y_t = y_{t-k}$
- $\phi(B) = 1 - \phi_1 B - \dots - \phi_p B^p$  is the AR polynomial
- $\theta(B) = 1 + \theta_1 B + \dots + \theta_q B^q$  is the MA polynomial
- $\epsilon_t$  is a white noise error term

Once the model is fitted using maximum likelihood estimation (MLE), future wind speeds  $\hat{y}_{t+h}$  are forecasted recursively:

$$\hat{y}_{t+h} = \sum_{i=1}^p \phi_i \hat{y}_{t+h-i} + \sum_{j=1}^q \theta_j \hat{\epsilon}_{t+h-j} \quad (17.2)$$

## 17.2 LSTM Network Formulation

LSTM is a type of recurrent neural network (RNN) suitable for capturing long-term dependencies in sequential data. The LSTM unit at time  $t$  is defined by the following equations:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad (17.3)$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad (17.4)$$

$$\tilde{C}_t = \tanh(W_C x_t + U_C h_{t-1} + b_C) \quad (17.5)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad (17.6)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad (17.7)$$

$$h_t = o_t \odot \tanh(C_t) \quad (17.8)$$

The forecasted wind speed is given by:

$$\hat{y}_{t+1} = W_y h_t + b_y \quad (17.9)$$

## 17.3 Forecasting Pipeline

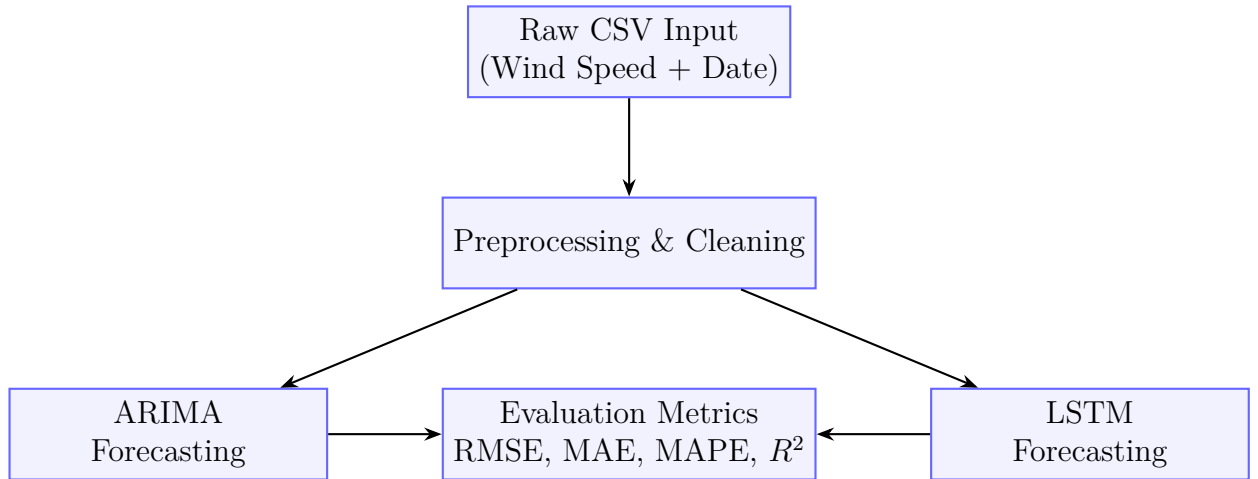


Figure 17.1: Model Forecasting Pipeline

## 17.4 Evaluation Metrics

Let  $y_t$  be the actual wind speed at time  $t$ , and  $\hat{y}_t$  be the predicted wind speed.

### 17.4.1 Root Mean Square Error (RMSE)

RMSE measures the square root of the average squared differences between predicted and actual values. It penalizes larger errors more heavily, providing insight into the model's overall accuracy.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{t=1}^n (y_t - \hat{y}_t)^2} \quad (17.10)$$

### 17.4.2 Mean Absolute Error (MAE)

MAE is the average of the absolute differences between predicted and actual values, reflecting the average magnitude of errors in the predictions without considering their direction.

$$\text{MAE} = \frac{1}{n} \sum_{t=1}^n |y_t - \hat{y}_t| \quad (17.11)$$

### 17.4.3 Mean Absolute Percentage Error (MAPE)

MAPE expresses the average absolute error as a percentage of the actual values, useful for understanding error relative to the scale of data.

$$\text{MAPE} = \frac{100}{n} \sum_{t=1}^n \left| \frac{y_t - \hat{y}_t}{y_t} \right| \quad (17.12)$$

### 17.4.4 Coefficient of Determination ( $R^2$ Score)

The  $R^2$  score quantifies the proportion of the variance in the actual values that is explained by the model. A value closer to 1 indicates a better fit.

$$R^2 = 1 - \frac{\sum_{t=1}^n (y_t - \hat{y}_t)^2}{\sum_{t=1}^n (y_t - \bar{y})^2} \quad (17.13)$$

where  $\bar{y}$  is the mean of the actual wind speeds.

## 17.5 Deployment Error Margin

Based on the observed metrics in our training pipeline (MAE  $\approx 11.1$ , RMSE  $\approx 17.3$ ), we permit a model deployment tolerance of up to **18% error**, consistent with established thresholds from wind speed forecasting.

### 17.5.1 Handling Missing Data: Forward-Backward Interpolation

In time series data, missing values can occur due to sensor errors or data transmission loss. To ensure the continuity and quality of the input data, forward-backward interpolation is applied to estimate missing values.

**Forward Interpolation:** The missing value at time  $t$  is approximated using the last known previous value:

$$\hat{y}_t^{(f)} = y_{t-1} \quad (17.14)$$

where  $\hat{y}_t^{(f)}$  is the forward-interpolated value at time  $t$ , and  $y_{t-1}$  is the last observed value before  $t$ .

**Backward Interpolation:** The missing value at time  $t$  is also approximated using the next known future value:

$$\hat{y}_t^{(b)} = y_{t+1} \quad (17.15)$$

where  $\hat{y}_t^{(b)}$  is the backward-interpolated value at time  $t$ , and  $y_{t+1}$  is the next observed value after  $t$ .

**Final Interpolated Value:** The final estimate is taken as the average of forward and backward interpolations, providing a balanced estimation:

$$\hat{y}_t = \frac{\hat{y}_t^{(f)} + \hat{y}_t^{(b)}}{2} = \frac{y_{t-1} + y_{t+1}}{2} \quad (17.16)$$

This approach assumes linearity between known points and is effective for small gaps in data, ensuring smoother input for forecasting models.

# 18 Bill of Materials

## 18.1 Material List and Hardware Bill of Materials

The following hardware components were used for developing, testing, and running the hurricane intensity prediction system.



Materials	Description	Qty	Link	Price
	Lenovo LOQ Gen 9 used for model training, analysis, and Streamlit UI.	1	<a href="#">Link</a>	591.75 €
	SanDisk 1TB SSD for storing datasets, models, and backups.	1	<a href="#">Link</a>	79.99 €

Table 18.1: Hardware Bill of Materials

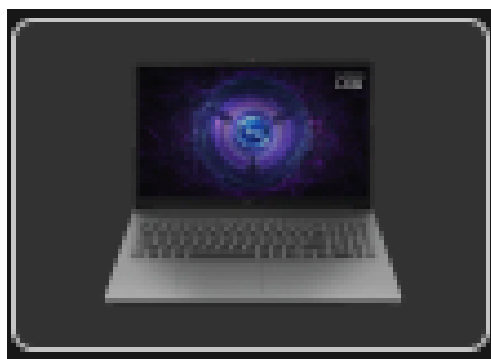


Figure 18.1: Lenovo LOQ Essential Gen 9



Figure 18.2: SanDisk 1TB SSD

## 18.2 Software Requirements and Python Libraries

Software / Library	Description	Platform / Interface	Cost
Python	Main language for development and modeling.	VS Code, Jupyter	Free
Streamlit	Web framework used to build a user-friendly UI for predictions.	Browser UI	Free
scikit-learn	For error metrics (MAE, RMSE), splitting, and preprocessing.	Python Library	Free
statsmodels	Used for ARIMA implementation and time series handling.	Python Library	Free
tensorflow + keras	Used for building and training LSTM models.	Python Library	Free
joblib	For saving and loading trained models (serialization).	Python Utility	Free
matplotlib, plotly	For static and interactive visualization of trends and predictions.	Python Visualization Tools	Free
pandas, numpy	For data cleaning, manipulation, time series indexing, and numerical ops.	Python Core Libraries	Free
requests	Enables HTTP requests (for future API support).	Python Library	Free
LaTeX	Used to write this manual.	TeXstudio	Free

Table 18.2: Software and Libraries Used

## 18.3 requirements.txt

The following Python packages were essential for executing the entire hurricane intensity forecasting pipeline:

Listing 18.1: requirements.txt

```
streamlit
pandas
numpy
joblib
keras
tensorflow
statsmodels==0.13.5
scikit-learn
matplotlib
plotly
requests
```

## 18.4 Explanation of Key Libraries

- **pandas**, **numpy** – Handle dataset loading, cleaning, transformations, and numerical operations.
- **scikit-learn** – For calculating evaluation metrics (e.g., MAE, RMSE), scaling, and data preprocessing.
- **statsmodels** – Implements the ARIMA time series forecasting algorithm.
- **keras** + **tensorflow** – Used to build, train, and validate the LSTM deep learning model.
- **matplotlib** + **plotly** – Generate side-by-side visual comparisons of predicted vs actual hurricane intensities.
- **streamlit** – Provides a clean web interface where users can upload data and view model outputs interactively.
- **joblib** – Saves and reloads model states, enabling fast deployment and reproducibility.
- **requests** – Facilitates web-based data interactions (for future integration of APIs).

# 19 Given documentation

## 19.1 Use of Best Practice Document

In developing our project, Hurricane Intensity Prediction Using ARIMA Model, we followed a structured best practices framework to ensure clarity, reproducibility, and maintainability. This included setting up a logically organized directory layout and applying uniform naming conventions (e.g., hurricane\_data.csv, README.md, requirements.txt) for all files. We configured essential tools such as Git for version control, and LaTeX for preparing clean documentation.

Our approach began with a thorough literature review focusing on time series forecasting models, particularly the ARIMA model, informed by academic papers, technical blogs, and authoritative sources. We created a project timeline defining milestones, deliverables, and roles to ensure consistent progress and adherence to documentation standards.

All written materials—including reports, charts, and tables—were prepared with attention to detail, ensuring proper labeling and citation. This rigor reflected the standards set forth in the best practice guidelines.

Version control practices were strictly followed by maintaining detailed Git commit histories, enabling full traceability of experiments and data preprocessing steps. Documentation was synchronized with code updates, making it possible to trace each figure and result directly back to the source code.

## 19.2 Same Notation

Notation	Meaning
ARIMA	Auto-Regressive Integrated Moving Average
CSV	Comma-Separated Values
RMSE	Root Mean Squared Error

## 19.3 Supplements

In addition to the best practices document, we referenced several additional resources to support our work:

- Official documentation and tutorials on the ARIMA model, particularly from statistical libraries such as statsmodels.
- Blogs and community guides for Python environment setup and data analysis workflows.
- Metadata and discussions from the NOAA hurricane dataset page on Kaggle, which clarified data coverage and handling of missing values.



# 20 Improvements

This section discusses possible future improvements to enhance the hurricane prediction system. Some are minor enhancements, while others are major upgrades that can improve accuracy, usability, and performance.

## 20.1 Minor Changes

These are small but effective updates that can be applied easily to make the system better.

### 20.1.1 List of Minor Changes

- Add tool tips and help texts in the user interface to guide users.
- Improve file name handling to prevent errors when saving results.
- Add more color options and themes to the forecast plots.
- Include unit tests for each function to check correctness.
- Add better error messages for wrong file types or missing data.

## 20.2 Major Improvements

These are significant upgrades that may require more development effort but have the potential to substantially improve the system.

### 20.2.1 Documentation of the Major Improvements

1. **Model Expansion**  
Future versions can include additional models such as Prophet, XGBoost, and ensemble methods. Prophet handles seasonality well, XGBoost is known for speed and accuracy, and ensemble models can boost robustness. [CM+24]
2. **Data Augmentation**  
Currently, the model only uses wind speed. Including additional features such as pressure, humidity, storm category, and sea surface temperature may improve prediction accuracy. [Ema05]
3. **Hyperparameter Tuning**  
Instead of setting parameters manually, automated techniques like Grid Search or Bayesian Optimization can be used to find optimal settings efficiently.
4. **Visualization Enhancements**  
Replace static forecast plots with interactive dashboards using tools like Streamlit, Dash, or Plotly. This would allow users to zoom, filter dates, and compare storm tracks.

**5. Mobile-Friendly Version**

Design a Progressive Web App (PWA) or simplified mobile interface for field teams and emergency responders to access predictions on mobile devices.

**6. Alert System Integration**

Integrate an alerting system that sends notifications (via email or SMS) when a severe storm is predicted. This supports emergency preparedness and disaster response. [Fed24]

# Bibliography

- [Aba+16] M. Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *arXiv preprint arXiv:1605.08695* (2016). URL: <https://arxiv.org/abs/1605.08695>.
- [BJR15] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 5th. Hoboken, NJ: Wiley, 2015. ISBN: 9781118675021.
- [Box+15] G. E. Box et al. *Time Series Analysis: Forecasting and Control*. 5th ed. Preview available on Google Books. Wiley, 2015. ISBN: 9781118675021. URL: <https://books.google.com/books?id=6JlfBQAAQBAJ>.
- [CM+24] R. Ciriminna, F. Meneguzzo, et al. “An Ensemble Approach to Predict a Sustainable Energy Plan for London Households”. In: *Sustainability* 17.2 (2024). DOI: ....
- [Cha+00] P. Chapman et al. “CRISP-DM 1.0: Step-by-step data mining guide”. In: 2000. URL: <https://www.the-modeling-agency.com/crisp-dm.pdf>.
- [DK+19] M. DeMaria, J. A. Knaff, et al. “Evaluation of Hurricane Intensity Forecast Models”. In: *Monthly Weather Review* 147.10 (2019), pp. 3621–3637.
- [DP13] X. Dong and D. Pi. “Novel method for hurricane trajectory prediction based on data mining”. In: *Nat. Hazards Earth Syst. Sci.* 13 (2013), pp. 3211–3220. DOI: [10.5194/nheiss-13-3211-2013](https://nheiss.copernicus.org/articles/13/3211/2013/nheiss-13-3211-2013.pdf). URL: <https://nheiss.copernicus.org/articles/13/3211/2013/nheiss-13-3211-2013.pdf>.
- [Ema03] K. Emanuel. “Tropical cyclones”. In: *Annual Review of Earth and Planetary Sciences* 31.1 (2003), pp. 75–104.
- [Ema05] K. Emanuel. *Divine Wind: The History and Science of Hurricanes*. New York: Oxford University Press, 2005. ISBN: 9780195149418.
- [FPSS96] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. “From Data Mining to Knowledge Discovery in Databases”. In: *AI Magazine* 17.3 (1996), pp. 37–54. DOI: [10.1609/aimag.v17i3.1230](https://doi.org/10.1609/aimag.v17i3.1230).

- 
- [Fed24] Federal Emergency Management Agency. *Integrated Public Alert & Warning System (IPAWS)*. <https://www.fema.gov/emergency-managers/practitioners/integrated-public-alert-warning-system>. Accessed: 2024-06-09. 2024.
- [HA21] R. J. Hyndman and G. Athanasopoulos. *Forecasting: Principles and Practice*. 3rd. OTexts, 2021.
- [HS97] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.
- [Har+20] C. R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: <https://www.nature.com/articles/s41586-020-2649-2.pdf>.
- [Hun07] J. D. Hunter. “Matplotlib: A 2D Graphics Environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55. URL: <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4160265>.
- [Inc23] S. I. Inc. *SAS 2023 Annual Report and Corporate Overview*. 2023. URL: <https://www.sas.com/content/dam/SAS/documents/corporate-collateral/annual-report/company-overview-report.pdf>.
- [KD03] J. Kaplan and M. DeMaria. “Large increases in the number of intense Atlantic hurricanes during the past four decades”. In: *Science* 293.5529 (2003), pp. 474–479.
- [Kag21] Kaggle. *Atlantic Hurricane Dataset (1975–2021)*. Available at: <https://www.kaggle.com/datasets/>. 2021.
- [Kor+16] M. Kordmahalleh et al. “A Sparse Recurrent Neural Network for Trajectory Prediction of Hurricanes”. In: *Proceedings of the 4th International Conference on Big Data and Smart Computing (BigComp)*. Also presented at Climate Change AI ICML Workshop. 2016. URL: <https://s3.us-east-1.amazonaws.com/climate-change-ai/papers/icml2021/38/paper.pdf>.
- [Lak+07] V. Lakshmanan et al. “The Warning Decision Support System–Integrated Information (WDSS-II)”. In: *Weather and Forecasting* 22.3 (2007), pp. 596–612. DOI: 10.1175/WAF1009.1. URL: [https://www.weather.gov/media/mdl/LakshmananStumpf2007\\_WDSS.pdf](https://www.weather.gov/media/mdl/LakshmananStumpf2007_WDSS.pdf).

- [Li+17] Y. Li et al. “Leveraging LSTM for rapid intensifications prediction of tropical cyclones”. In: *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* IV-4/W2 (2017), pp. 101–105. DOI: [10.5194/isprs-annals-IV-4-W2-101-2017](https://doi.org/10.5194/isprs-annals-IV-4-W2-101-2017). URL: <https://isprs-annals.copernicus.org/articles/IV-4-W2/101/2017/>.
- [Li+20] Z. Li et al. “A Long Short-Term Memory Model for Global Rapid Intensification Prediction”. In: *Weather and Forecasting* 35.4 (2020), pp. 997–1013. DOI: [10.1175/WAF-D-19-0199.1](https://doi.org/10.1175/WAF-D-19-0199.1). URL: <https://journals.ametsoc.org/view/journals/wefo/35/4/wafD190199.xml>.
- [McK11] W. McKinney. “Pandas: a foundational Python library for data analysis and statistics”. In: *Proceedings of the 9th Python in Science Conference*. 2011, pp. 51–56. URL: <https://conference.scipy.org/proceedings/scipy2011/mckinney.html>.
- [ON21] N. Oceanic and A. A. (NOAA). *HURDAT2: Atlantic Hurricane Database*. Available at: <https://www.nhc.noaa.gov/data/>. 2021.
- [PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: <https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf>.
- [Pyt25] Python Software Foundation. *Python Package Index*. <https://pypi.org/>. [Online; accessed 19-June-2025]. 2025.
- [RJ17] M. A. Russell and J. Jurney. *Agile Data Science 2.0: Building Full-Stack Data Analytics Applications with Spark*. Sebastopol, CA: O’Reilly Media, 2017. ISBN: 9781491960110.
- [SP10] S. Seabold and J. Perktold. “Statsmodels: Econometric and Statistical Modeling with Python”. In: *Proceedings of the 9th Python in Science Conference*. 2010, pp. 57–61. URL: <https://proceedings.scipy.org/articles/Majora-92bf1922-011>.
- [Scu+15] D. Sculley et al. “Hidden Technical Debt in Machine Learning Systems”. In: *Advances in Neural Information Processing Systems* 28 (2015). URL: <https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf>.

- 
- [Shi+15] X. Shi et al. “Convolutional LSTM network: A machine learning approach for precipitation nowcasting”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2015, pp. 802–810. URL: <https://proceedings.neurips.cc/paper/5955-convolutional-lstm-network-a-machine-learning-approach-for-precipitation-nowcasting.pdf>.
- [Sou+20] D. B. C. de Sousa et al. “Studying the evolution of exception handling anti-patterns in a long-lived large-scale project”. In: *Journal of the Brazilian Computer Society* 26.1 (2020), pp. 1–16. DOI: [10.1186/s13173-019-0095-5](https://doi.org/10.1186/s13173-019-0095-5). URL: <https://doi.org/10.1186/s13173-019-0095-5>.
- [Su+11] Y. Su et al. “A New Data Mining Model for Hurricane Intensity Prediction”. In: *Proceedings of the International Conference on Data Mining (ICDM)*. Available at: <https://s2.smu.edu/~mhd/7331f10/IntensityPrediction.pdf> [Accessed: 2025-06-05]. IEEE. Dallas, Texas, 2011.
- [Tiw+22] D. Tiwari et al. “Production Monitoring to Improve Test Suites”. In: *IEEE Transactions on Reliability* 71.3 (2022), pp. 1381–1397. DOI: [10.1109/TR.2021.3101318](https://doi.org/10.1109/TR.2021.3101318). URL: <https://arxiv.org/pdf/2012.01198>.
- [WB00] R. Wirth and M. Bramer. “CRISP-DM: Towards a Standard Process Model for Data Mining”. In: *Proceedings of the Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining*. Manchester, UK, 2000, pp. 29–39.
- [ZPH01] G. Zhang, B. Patuwo, and M. Hu. “Forecasting with artificial neural networks: The state of the art”. In: *International Journal of Forecasting* 14.1 (2001), pp. 35–62. DOI: [10.1016/S0169-2070\(99\)00016-4](https://doi.org/10.1016/S0169-2070(99)00016-4). URL: [https://www.oscogen.ethz.ch/members/literature\\_restricted%20access/ann\\_for\\_001.pdf](https://www.oscogen.ethz.ch/members/literature_restricted%20access/ann_for_001.pdf).
- [Zha+22] Y. Zhang et al. “A Short-Term Tropical Cyclone Intensity Forecasting Method Based on BHT-ARIMA”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 4. 2022, pp. 4092–4100. DOI: [10.1609/aaai.v36i4.20394](https://doi.org/10.1609/aaai.v36i4.20394). URL: <https://ojs.aaai.org/index.php/AAAI/article/view/20394>.