DISSERTATION

A FRAMEWORK FOR DETECTING DEFECTS IN SEQUENTIAL INPUTS TO MODELING AND SIMULATION USING MACHINE LEARNING TECHNIQUES

Submitted by

Nathaniel R. Brown

Department of Systems Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2025

Doctoral Committee:

Advisor: Steven Simske

Sudipto Ghosh
Chandrasekaran Venkatachalam
Ali Pezeshki

ABSTRACT

A FRAMEWORK FOR DETECTING DEFECTS IN SEQUENTIAL INPUTS TO MODELING AND SIMULATION USING MACHINE LEARNING TECHNIQUES

This work presents a method for detecting defects in sequential data inputs for digital twins (DT) during simulations, emphasizing the importance of input validation for ensuring the accuracy and reliability of the simulation results. By thoroughly validating input data, researchers and practitioners can have confidence in the validity of their models, ultimately leading to better decision-making processes and outcomes that are more successful. As DTs continue to expand in complexity, there are an increasing number of mechanisms that may produce undesirable output. An external data stream is just one such potential source of faulty DT behavior and must be analyzed during simulation execution. The proposed framework for validating inputs in real time offers a way to improve the quality and credibility of DTs, guiding future research in the evolving field of modeling and simulation. The case study described in this paper involves using second-order polynomial regression, autoregressive integrated moving average (ARIMA), artificial neural networks (ANN), and a meta-algorithm to detect defects in rocket trajectory data streams, highlighting the effectiveness of validation techniques. This method shows promise, as it successfully identified defects in trajectories in real time using only historical data without knowledge of the future of the data stream. The novelties in this work include 1) using machine learning to validate DT inputs, and 2) performing this validation on sequential data in real time to protect modeled results. This research contributes valuable insights to the field, emphasizing the significance of input validation for enhancing the quality and accuracy of simulation models.

ACKNOWLEDGEMENTS

To my advisor, Dr. Steve Simske, for his patience, guidance, and mentorship through this adventure. To Drs. Ghosh, Venkatachalam, and Pezeshki in my graduate committee for their time and recommendations. To my wife, Amy, whose love and encouragement made this journey possible. To my children, David and Emily, may your endless curiosity continue to inspire and lead you to discover the extraordinary. To my parents for their support and encouragement over the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

Modeling and simulation (M&S) has emerged as a powerful tool for understanding complex systems, allowing researchers and practitioners to investigate and predict outcomes in a controlled environment. By creating virtual replicas of real-world systems, these tools enable scientists to gain insights and test hypotheses that would otherwise be unfeasible or resource intensive to explore directly. However, the quality of a model's output is not higher than the quality of its input. This work aims to explore the impact that inputs have on M&S systems, the challenges that M&S developers and users face when sanitizing inputs, and propose a framework for validating sequential inputs on the fly during run time. Through an in-depth examination of realistic data using advancements in machine learning, this research seeks to contribute to the existing body of knowledge, guiding future researchers towards new frontiers in this rapidly evolving field.

An advanced form of M&S is the concept of a digital twin (DT). A DT is a digital model of a real-world system designed to be indistinguishable from its physical counterpart for all intended purposes [1]. It is a representation of a process that may be used in lieu of the real system to understand it better or to supplement real data with modeled data. It is frequently cheaper and easier to use a model for these purposes rather than using up the real system's resources to generate data or make changes to it for experimentation. Depending on the system, stress tests in particular are expensive, as they may require destruction of components, a consequence that a model may not face.

The intention is to execute the DT with given inputs to record and observe an output. Some of these inputs could be parameters that are set before a run and others are fed into the simulation live. Outputs are often used for performance analysis, testing assumptions about the system, or for comparison with the real system output to verify and validate the model. DTs in particular

are advantageous in that the number of experimental executions is only limited by computational resources [2].

George Box famously said, "All models are wrong, but some are useful" [3]. This phrase was originally used by Box in reference to statistical models, but has been applied more recently to DTs in science and engineering [4]. The phrase suggests that, while no DT can perfectly represent reality, some DTs can still provide useful insights or assistance in understanding a real system. In other words, DTs are simplified representations or approximations of reality, and while they may not be entirely accurate, they can still provide value in aiding decisions or gaining understanding. Other uses of DTs include training, system performance analysis, and the development and testing of real systems.

DTs are increasingly valuable tools during the concept development and design phases of a system [5]. Their advantage lies in their ability to serve as detailed models that emulate the behavior and characteristics of a physical system. During the early development stages, DTs facilitate rapid iteration and exploration of design options without the need for costly physical prototypes. One of the key strengths of DTs in this context is their utility for digital prototyping. By creating a digital replica, practitioners can assess how different design choices impact system performance, functionality, and safety. This allows testing of various scenarios, identifying potential issues, and refining requirements before moving to physical prototyping and manufacturing. Additionally, DTs enable high-level validation of system architectures, ensuring that fundamental design objectives are feasible and aligned with an organization's goals.

A practical example of using DTs as a prototype is seen in the early design phase of an aircraft. In this case, the DT would incorporate known physical effects simulated against a digital model of the aircraft. These simulations can include airflow patterns and material stress distributions, providing insight into how the actual aircraft might perform under different conditions. Such a DT prototype allows designers to assess the impact of various design choices such as wing shape and materials without constructing physical prototypes, significantly reducing development time and costs.

A DT can enable performance assessment of a real-world system through simulation. By mirroring the operational capabilities of the physical system, the DT allows practitioners to monitor performance metrics, identify anomalies, and predict potential failure points before they occur. The DT acts as a valuable tool to assess, diagnose, and improve the performance of complex systems in a cost-effective manner.

Another application of DTs is to train users of a system. By creating a virtual replica of a real-world system, DTs offer a safe, controlled environment where users can gain practical experience without the risks associated with operating the actual system [6]. For example, in high-risk jobs like air traffic control, training on a DT allows controllers to practice complex scenarios, their response to emergencies, and refine their decision-making skills in a realistic setting where they can fail with no harm done. Simulated training environments enable users to familiarize themselves with system behaviors, interfaces, and responses under various conditions, increasing confidence before using the real system.

In addition to developing and testing the modeled system, DTs can be used to develop and test other associated systems. Most systems do not operate in a vacuum and may interact with others in a complex system of systems environment. A DT may serve as a stand-in for a real-world element to test adjacent systems by providing realistic data and interactions. This is a benefit because users do not need to bring down a real system or build a separate, expensive test-only system for this purpose and the adjacent elements have the advantage of a realistic test environment.

A commonly heard phrase in the M&S world is "garbage in, garbage out", meaning that bad outputs follow from bad inputs. Data from a simulation cannot be trusted if the configuration has defects. What decisions should be made on such data? It is important that results are trusted to obtain the benefits described above. Consequences for defective input data are numerous and include negative training, poor performance analysis of the real system, and stalled or deficient development [5, 7]. Negative training is training that occurs using a DT element that inadequately emulates the real system's behavior. Using a faulty DT element can also stall the development of the real system through faulty performance analysis results.

DT inputs may take many forms, such as parameters or stimulation data that are configured prior to execution. Stimulation inputs are data that stimulate the DT to react. An example could be trajectories that are read into the DT prior to execution and then passed through to subsystems in the model for reaction. Parameters are other types of input that change behavior in the DT, such as the position of a modeled radar. These sorts of data could be simpler to validate due to understanding the limitations of the input. For example, one may assume that the latitude of a position should be some value between -90 and 90 degrees, indicating how far north or south of the equator the radar should be modeled. Understanding this limitation means that the DT can catch bad data before running the simulation.

There are advantages to a parameterized model, including flexibility, control, and the ability to introduce random effects [8]. Sufficient parameter options give the flexibility to test new conditions or control for numerous known conditions. The ability to introduce random effects may be advantageous for performance testing, verifying the real system, or stress testing.

In addition to pre-execution inputs, some DT environments support live, sequential inputs. Instead of reading in trajectories before a run, a DT may support accepting live inputs in real time. This is beneficial in a digital twin environment where several elements are interacting with each other and it is simpler to configure one data provider that sends the data to all elements at the same time rather than configure each element individually to read in the data. An example is an object trajectory provider sending trajectories live during a simulation run to several elements, each of which then reacts to the data.

The Institute of Electrical and Electronics Engineers (IEEE) develops and publishes the Distributed Interactive Simulation (DIS) standard, which provides real-time data to multiple computers during M&S events [9]. The DIS standard is used worldwide, especially by the United States Department of Defense (DoD), to conduct war games and exercises. Continuing the trajectory example, the DIS protocol data units (PDUs) can be used to provide sequential data in real time that contain a trajectory state to M&S elements. In particular, Entity State PDUs contain an object's current state, including its location, velocity, and other relevant information (see Table 1.1).

Table 1.1: DIS Entity State PDU Contents.

Field Size (bits)	**Entity State PDU Fields**
64	Entity Type
64	Alternate Entity Type
96	Entity Linear Velocity
192	Entity Location
96	Entity Orientation
32	Entity Appearance
320	Dead Reckoning Parameters
96	Entity Markings
32	Capabilities
128	Variable Parameter Records (may be repeated)

DIS Entity State PDUs are generally sent every five seconds for each object [9]. If the dead reckoning (state estimation) algorithm determines that the difference between the dead reckoning state and the actual state is greater than a predetermined threshold, more frequent Entity State PDU updates are sent. This feature reduces communication processing.

A DIS provider is a software tool that ingests data sources, transforms them into DIS PDUs, and sends them to the elements of a federated M&S system across a network. While these software tools are tested, there may still be deficiencies that introduce defects. If there is a defect in the source files, a bug in the parser and translator, or an issue with the network, the data could be corrupted. Some corruptions, such as fields filled in with not-a-number (NaN) or infinite (Inf), are trivial to catch, but others such as defects to a trajectory state are more difficult to detect and are explored here.

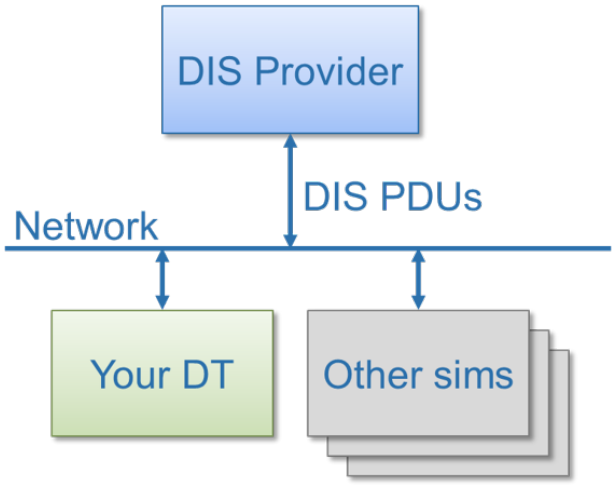

Figure 1.1: Federated network of DTs with a DIS provider.

Often a DIS provider is one part of a federated network of DTs, as in Figure 1.1. M&S elements receive Entity State PDUs to update objects on internal data structures, sometimes called a gameboard. This so-called truth trajectory data is used to stimulate a model. For example, a radar model may use the truth trajectory combined with radar cross section (RCS) data to simulate a radar return and determine whether an object is detected. Sensitive systems, and thus models, are more susceptible to defective data and may produce unreliable results when defects are accepted as valid.

The case study described in this paper utilizes a set of rocket trajectories to test a defect detection model. A DT consuming trajectory data during simulation execution could read in the data directly or receive it live from an external data source, such as a DIS PDU provider. Live sequential data from external sources present a challenge to DTs, as they cannot verify the veracity of the data prior to execution, and data defects may introduce faulty DT modeling. The case study applies second-order polynomial regression, autoregressive integrated moving average (ARIMA), artificial neural networks (ANN), and meta-algorithms to rocket trajectories with defects to test the ability to detect these defects in real time. Using these techniques, this work demonstrates a step toward robust data input management in DT systems.

1.2 Problem Statement

Since the quality of the DT output is not higher than the quality of its inputs, defective inputs spoil the results. This work focuses on validating sequential inputs during simulation execution. Sequential data are information in a particular order, such as the trajectory of a rocket over time. Real-time simulation refers to a DT or a system of DTs that are executed such that the rate of time simulated matches real time, such as on a clock. This work does not differentiate between real-time and non-real-time simulation data validation, as the problem remains the same in either case. Additionally, this work does not cover the verification of inputs prior to simulation execution since these values are generally controlled directly by user input and may be reviewed before execution.

Using the example of DIS Entity State PDUs, which are sequential inputs, there are a couple of problems that may arise during execution due to bugs in the software or input files [9]. The first is that the DIS software may fail to calculate an object's state correctly and send Entity State PDUs with fields that are considered "not a number" (NaN) or infinite (Inf). These field values are trivial to filter out for a DT and the PDU will typically be excluded as an input. The second problem, which this work focuses on in the case study, is a state update that appears valid on the surface, but contains defective data such as a location field with inaccurate altitude.

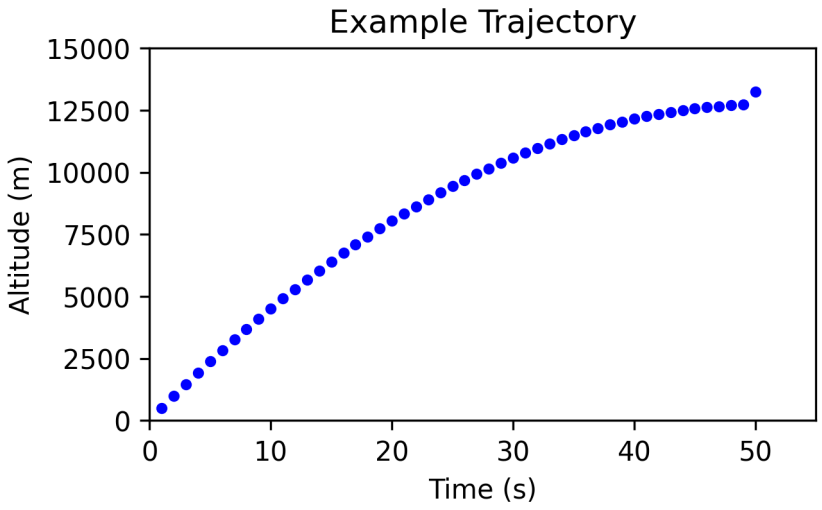

Figure 1.2: A trajectory provided by DIS Entity State PDU updates with a questionable update provided at simulation time 50.

Figure 1.2 shows a simple example trajectory's altitude that could have been provided by Entity State PDUs. The trajectory follows a smooth curve until a questionable state is provided at simulation time 50. Since only the history of the object is known, not the future, this deviation could be the result of a defect, or it is a valid state and the trajectory is changing.

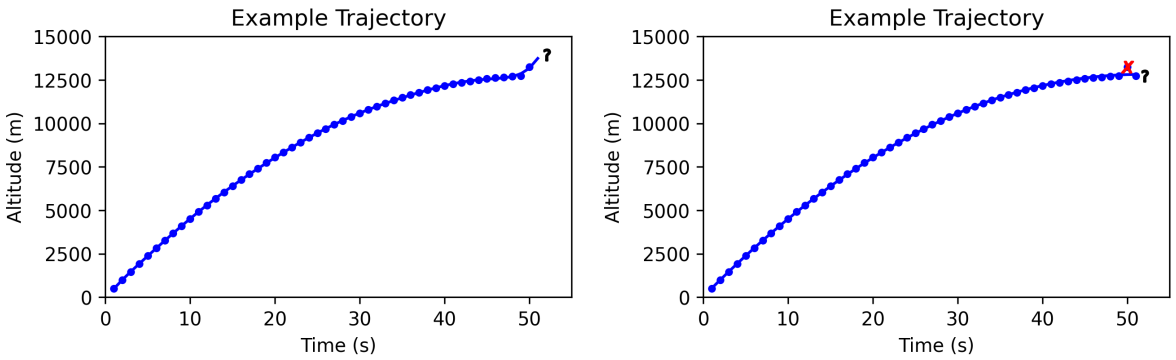

Figure 1.3: Two possibilities for the seemingly anomalous update: the update was valid (left) or a defect (right).

In Figure 1.3, the two possible results are shown. The first example demonstrates that the trajectory is changing and that the questionable state was valid. In this case, the DT should use the state update as a valid input. The second example shows that the questionable state was likely a defect and that state should have been ignored and flagged for review. Treating this defective state update as valid could result in negative impacts on DT performance. The consequences of poor DT performance include negative training, inaccurate model performance analysis, and flawed real system development.

Defective inputs can negatively influence the performance assessment of a DT. Performance assessment, or performance analysis, of a DT is the evaluation of the accuracy of a model compared to a real system. This analysis is an important step for the verification and validation (V&V) of a DT for other use cases [10]. If a perfect DT of a real system were to exist, even it would fail to provide useful output if it were provided defective inputs.

Negative training occurs when the outputs of a DT do not match real-world expectations and are used to train people to use a system [7]. If, for example, a radar model that is used to train air traffic

controllers outperforms the real radar, the trainees will be educated to expect better performance than what they will see in the field. Conversely, if the DT performs poorly, trainees will be led to believe that the real system is not as capable as in reality. In either case, negative training using poor DT output creates a false impression in those who use the model to train to a real system.

DTs are not only used to emulate an existing state of a real-world system, they can be a part of a development feedback loop to create or update a real system [11]. Some DTs may be used to quickly and inexpensively experiment with candidate configurations or upgrades that may be introduced to the real system. As in other use cases, these experiments require reliable outputs and are spoiled by bad data ingested by the DT.

1.3 Purpose of the Study

Since digital twins are useful tools, it is important that the results are correct. One step in this process is validating inputs, which include pre-execution inputs that may be verified before run time and sequential inputs that can only be validated during execution. The purpose of this study is to demonstrate that machine learning techniques are well suited to identify defects in sequential data inputs in real time during execution.

This study aims to answer the following three questions:

1. Can sequential input data with defects be detected in real time using second-order polynomial regression, ARIMA, and artificial neural networks?

2. Do these defect detection techniques have better performance working together in a meta-algorithm?

3. Are these techniques applicable to practitioners in the real world?

1.3.1 Research Questions

Research Question 1: Can sequential input data with defects be detected in real time using second-order polynomial regression, ARIMA, and artificial neural networks?

A positive answer to this question signifies a significant advancement in the field of real-time data monitoring and anomaly detection in DTs. It indicates that regression, ARIMA, and ANNs are capable of effectively identifying defects or irregularities in sequential input data as they occur when only the history of the data stream is known. This capability is crucial in applications where immediate detection can prevent failures, reduce downtime, or improve safety. The ability to detect defects in real time ensures that practitioners can respond quickly, minimizing potential damage.

The successful application of these models to defect detection demonstrates their robustness and adaptability across various types of data and environments. Second-order polynomial regression, with its capacity to model quadratic relationships, can capture non-linear patterns associated with anomalies. ARIMA models excel in modeling and forecasting time series data, enabling the detection of deviations from expectation. Artificial neural networks possess high flexibility and can learn complex non-linear relationships, making them powerful tools for identifying subtle or previously unseen defects. A positive result indicates that these models can be integrated into real-time systems and DTs, providing reliable insights.

Furthermore, a positive answer to this question encourages further research and development in the domain of real-time analytics. It promotes the effectiveness of combining traditional statistical methods with advanced machine learning techniques for defect detection tasks. This success can inspire new hybrid models, improved algorithms, and more sophisticated data processing techniques, pushing the boundaries of what is achievable in real-time modeling and simulation. In summary, it signifies a meaningful step towards more intelligent, responsive, and reliable DTs capable of maintaining high standards of quality in various industries.

Research Question 2: Do these defect detection techniques have better performance working together in a meta-algorithm?

A positive answer to this question signifies a meaningful advancement in the field of defect detection in sequential real-time data streams that serve as input for DTs. It suggests that integrating multiple techniques into a meta-algorithm can leverage the strengths of each individual

method, leading to improved accuracy, robustness, and reliability. This means a meta-algorithm could be used to effectively identify defects across diverse scenarios, reducing false positives and false negatives, which are critical metrics in quality control processes.

The significance extends to a deeper understanding of the complementary nature of different defect detection techniques. When the three primary methods are combined in a meta-algorithm, they can compensate for each other's limitations, such as varying sensitivities to different defect types or conditions. This collaboration can enhance the detection of subtle or complex defects that might be missed when using the techniques independently. Consequently, practitioners can achieve higher standards of DT integrity that are flexible and scalable. This can lead to cost savings, faster DT execution times, and reduced reliance on manual inspection, ultimately streamlining DT use and analysis.

In summary, a positive answer to this question highlights the potential for collaborative, multi-technique approaches to significantly enhance defect detection capabilities. It signifies progress towards more accurate, efficient, and adaptable defect detection systems, which are vital for maintaining high standards in modeling and simulation.

Research Question 3: Are these techniques applicable to practitioners in the real world?

A positive answer to the question, which relies on the success of the previous two research questions, signifies that the techniques used in this study to detect defects in sequential data are indeed practical and relevant for real-world practitioners. This success will underscore the effectiveness of these methods in environments where timely identification of anomalies can prevent failures, reduce downtime, and improve the overall reliability of DTs. Practitioners can confidently adopt these techniques, knowing that they are grounded in solutions that have been proven to work. This positive outcome indicates that these techniques are accessible and adaptable to the complexities of real-world data streams.

Practitioners are more likely to invest resources into the implementation of these techniques when they are proven to be applicable in practice. This can lead to the development of more advanced solutions that address specific needs, supporting a feedback cycle of continuous improvement and technological advancement in defect detection processes. Additionally, the applicability of these techniques in real-world scenarios highlights the importance of bridging the gap between research and practice. It emphasizes that theoretical developments are not merely academic exercises, but have benefits when translated into tools for practitioners to use.

In summary, a positive answer to this question confirms that the defect detection techniques for DT sequential data inputs are not only sound but also valuable. It reassures practitioners of their usability in DTs and other applications, encourages their use, and supports ongoing advancements that can lead to more reliable DTs and systems across various domains.

Chapter 2

Review of Literature

2.1 Detecting Defects and Anomalies

The use of machine learning to detect defects and anomalies is well-established. Most applications using digital twins focus on post-execution analysis, not real-time validation [12–15]. However, some examples of real-time data validation include using autoencoders to detect false data injection attacks to thwart cyber threats [16], detecting anomalies in measurements using artificial neural networks to identify sensor faults [17], and validating DTs in real time alongside the real-world process [18].

A conference paper by Averill M. Law & Associates in 2022 presented a framework and techniques for building valid simulation models [12]. This work emphasized the importance of DTs by highlighting their benefits, such as cost and flexibility, and focuses on ensuring that DTs are valid. Law defines DT validity as a model close enough to its real-world counterpart that it can be used to make decisions similar to those that would be made if experimenting with the system itself. The paper points out that a DT does not need to be a perfect representation, it needs only approximate the actual system for intended use cases. It also mentions the cost of DT development as a factor in determining how high the fidelity of the model needs to be and that adding unnecessary capability comes with unnecessary cost. The paper does not mention machine learning as a technique, but is a comprehensive framework for validating simulation models after execution.

Khan *et al.* presents another paper detailing the importance of DT validation [13]. The paper indicates that incorrect simulation model behavior leads to improper decisions and can damage real equipment. This work focuses on synchronous simulation, where a real control system is connected to both a DT and a real production system. This concept is also used by Lugaresi *et al.*, which is discussed in Section 2.3 [18]. The DT is compared with the real system and is evaluated for deviations in real time. This creates a feedback loop where the DT is validated by real data and

the real system output is validated by the simulation model. This paper does not mention machine learning, but indicates that real-time analysis is critical to ensuring a validated model and system output.

Taking a step towards machine learning, Rebba *et al.* offers some statistical tools for validating simulation models [15]. This paper utilizes metrics, one based on Bayesian analysis and another on principal components analysis, to compare model prediction with experimental observation. This analysis was performed on data extracted from simulation post-execution rather than in real time.

One maturing field that is utilizing machine learning for real-time data analysis and anomaly detection is cybersecurity. Zidi *et al.* utilize support vector machines (SVM) to detect faults in wireless sensor networks [19]. In the paper, the authors compare SVM against other methods, such as Bayesian techniques and hidden Markov models, to demonstrate that the SVM is superior to the other methods in reducing false positives in their data. The data they used has been evaluated by other researchers using other techniques, which is a valuable concept in comparing defect detection methods.

Aboelwafa *et al.* have a paper detailing how to use machine learning to detect false data injection attacks [16]. It introduces a method using autoencoders to detect false data injection attacks in real time and discusses the cleaning of the data before it reaches its destination. The paper compares this technique against support vector machine approaches, such as those in papers listed above, and indicates that autoencoders may outperform that technique and are useful for recovering clean data from corrupted data.

Alippi investigated using clustering methods and hidden Markov models (HMM) to detect and isolate faults in cyber systems [20]. This research used a correlation-driven clustering method to group sensors according to specified characteristics and then applied an HMM to inspect data streams for faults. If a fault is detected, the data stream can be isolated for further impact analysis on the system.

Finally, in another application to cyber threat detection, Lee *et al.* applied various deep learning algorithms using artificial neural networks in an effort to decrease false positives while maintaining

high anomaly detection rates [21]. This work used two benchmark data sets to evaluate the performance of the methods compared to existing techniques. The goal was to use the model to detect cyber threats in real time by passing data streams through the model and evaluating the results. The technique controlled large-scale data by condensing it into events to be evaluated by the detection models, reducing computational intensity. It also appeared to utilize meta-learning methods by combining techniques internal to the model using fully connected neural networks, convolutional neural networks, and long short-term memory networks simultaneously.

2.2 Comparison of Machine Learning Techniques

There are a variety of machine learning techniques available that utilize supervised or unsupervised learning strategies to detect anomalies [22]. Some techniques used in the past include support vector machines [19, 23], hidden Markov models [20], and various artificial neural networks [21, 24]. Kalman filters are also used for this purpose [25, 26], though this paper uses Kalman filters for severity assignment (see 3.3). It seems prudent to start with simple models that are easy to train and explain and then move on to more complex, and possibly robust, techniques. Detection of defects relies on understanding the data, so it is expected that polynomial regression and ARIMA will be effective in supporting rocket trajectory modeling. Artificial neural networks are very adaptive to data, so it is expected that it will be a robust method that may not measure up to regression and ARIMA in this data, but could be used with greater success more generically.

With regression being the simplest model of the three primary techniques selected, it offers a host of benefits, including statistical interpretability and visualization of the model. Its simplicity may discourage practitioners from considering it for detecting defects and anomalies, but it can be a powerful tool with certain types of data. There are several types of regression model, including linear, polynomial, and logistic, so it can be used on a wide variety of data shapes. There is ongoing research in using regression to detect defects and outliers in the data [27–29]. With the increase in computational resources and the development of increasingly advanced models, practitioners should not forget about simpler models.

Polynomial regression's usefulness in this work is a direct result of the type of data used. Trajectories follow a natural parabolic curve when untouched by forces other than gravity, making the quadratic shape of second-order polynomial functions a clear choice to model trajectory behavior. Section 3.3.2 goes into greater detail about how this technique is used here, but the idea is that even with outside (or inside) forces acting on a trajectory such as wind, drag, and thrust, the trajectory is approximately parabolic at least locally. A kernel smoothing approach tests the local nature of the trajectory while predicting the next step.

ARIMA is hard to ignore when reviewing options for analysis of time series data. With its three components working in parallel, autoregression, moving average, and stationarity differencing, it is a powerful tool in time series forecasting. This robustness in forecasting time series data means that it can be used to compare its confidence in its forecast to actual data received as a defect detection mechanism. It has been used successfully in the past in a variety of domains and contexts [27, 30, 31]. ARIMA remains a useful tool, even as more complex techniques are developed.

Given the shape of the data, it is unlikely that ARIMA can significantly outperform regression. However, its flexibility gives it the ability to work with more general data and demonstrating that ARIMA works here indicates its viability to detect defects in other types of data streams [32]. It broadens the scope of data types that can be modeled and supports accurate forecasting in more complex situations. In particular, it may be used in situations where the general shape of the data is unknown, but this research will still use a statistical kernel to reduce the quantity of data that must be evaluated at every time step (see Section 3.3.3). This has the added benefit of feeding ARIMA data that is more locally smooth than would normally be given to an ARIMA model.

Increasing in complexity, artificial neural networks (ANN) make few assumptions about the data and will train the model to find any trends that are available [33, 34]. With the increasing computational resources available to researchers, ANN models are being trained to perform very complex tasks, including the use case of detecting anomalies [35–37]. There are many categories

of ANN, some of which are discussed in the following, but this research will use a single hidden layer approach.

Since the prior two primary methods are almost purpose-built for the task, ANN is not expected to significantly outperform either of them. The value of including this model in this research is to demonstrate that ANN is a powerful tool when detecting defects and that practitioners should consider using it to validate incoming data for their DTs. Section 3.3.4 details the method used in this study to train ANN models and use them to detect defects in sequential data in real time. This method holds promise in more general use cases, especially with data that do not conform to the assumptions of regression or ARIMA.

The prior three models were chosen to be explored in detail in this research and are explained more in other sections of this work. The next few models were not selected for analysis but need to be mentioned.

The seasonal-trend decomposition using LOESS (STL) technique is another model that can ingest noisy, generic data and extract valuable information. For STL, this comes in the form of a general trend for the data and analysis of its seasonality, if any exists (see Figure 2.1). The remaining information is residual differences from the average trend and seasonality. Although trend and seasonability are clearly valuable information, one of the uses of STL is to evaluate outliers in the residuals [38–40]. As it turns out, this powerful ability is best used after the data is collected as a look back and may not have much value in a sequential real-time analysis environment.

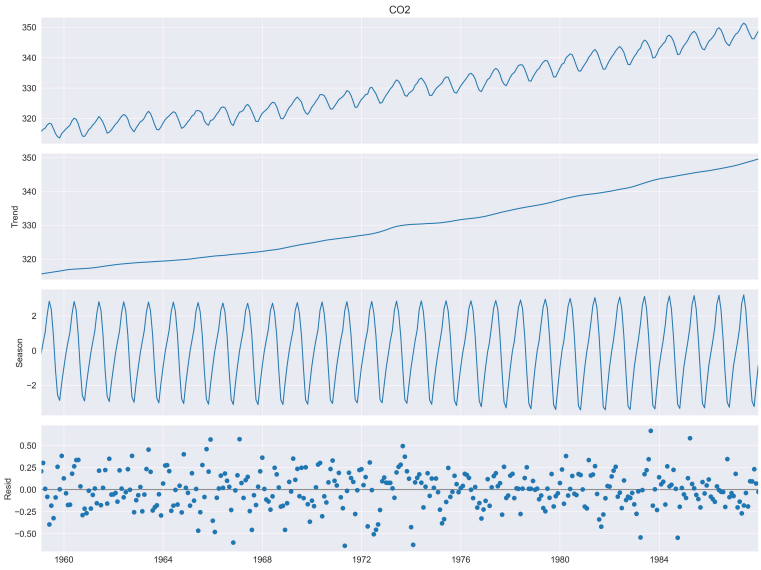

Figure 2.1: Example of how STL breaks down data into its seasonal and trend components.

The reason for this may be predictable to users of locally estimated scatterplot smoothing (LOESS). While LOESS can be a great predictor in the "center" of data's domain, there is significant statistical bias in the extremes of the domain. When the data are curved, especially at the tails, LOESS will often overestimate or underestimate the smoothed function. This leads to poor performance at the tails of the data, which is exactly where sequential, real-time data need the model to be most effective. In a quick test using the trajectory data used in this research, STL had a recall rate that was very low (<20%). Some of this poor performance was due to the wide confidence bounds at the tails of the domain, but mostly STL provides bad predictions when extrapolating outside the data domain, which is necessary for this analysis. This complication is visualized in Figure 2.2, where the LOESS estimate tends to overestimate the truth when the trend increases at the minimum of the domain and underestimate when the trend increases to the maximum.

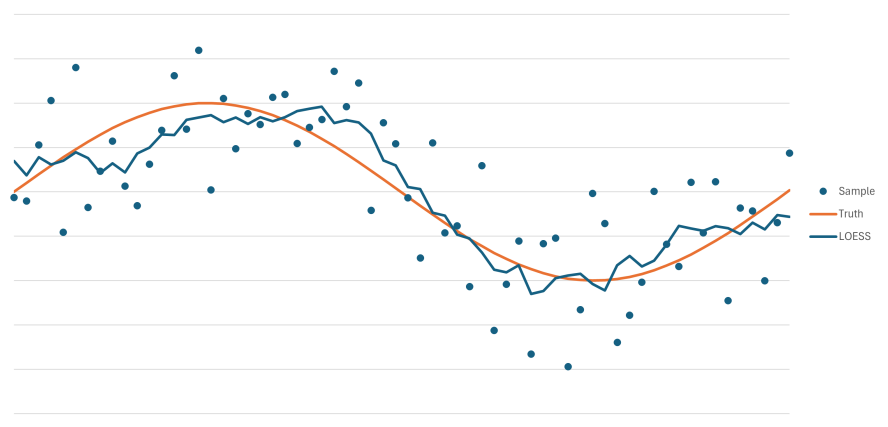

Figure 2.2: Example of how LOESS has a bias near the extremes of the data domain.

Another tool that is not analyzed in this work beyond mention is the long short-term memory (LSTM) network, which is a sequential learning method that predicts the output associated with the next time step and identifies anomalous data (see Figure 2.3) [41]. This method was designed for sequential data and keeps some memory of the past while continuously learning which data should be remembered or forgotten for its predictions. LSTM has been used extensively in defect and anomaly detection, especially as computational resources have increased, allowing for more complex models and better training [42–44]. The feedback loop built into LSTM models with the ability to forget some training results in a level of flexibility that is well suited for general use.

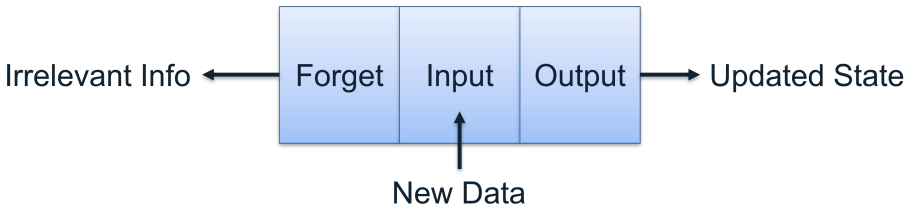

Figure 2.3: Basic LSTM workflow.

LSTM is so well suited to predicting sequential data that it is expected to perform very well in anomaly detection, especially with very predictable data such as the trajectories used in this case study. It is regrettable that there was not more time to evaluate LSTM in this work, but the

intention is to study LSTM in future work. It is almost certain to provide useful information and could be tuned to be computationally viable in a real-time DT environment.

Another machine learning technique that is used for anomaly detection is an autoencoder, which uses neural networks to learn and capture data patterns and structures. An autoencoder is used to detect defects and anomalies in the data by reconstructing those data and then comparing the reconstruction to its input to determine differences [45–47]. If the difference between the input (raw data) and output (reconstructed data) is great enough, it can be flagged as a defect. When using an autoencoder for anomaly detection, the threshold for the difference between input and output is a tuned value to balance recall and precision.

Autoencoders are often paired with other methods to train the model and reconstruct data. Combining the previous two techniques, an autoencoder based on an LSTM network can utilize the best qualities of each method on sequential data [44, 48, 49]. This approach leverages the autoencoder's ability to learn from normal data, making it effective for identifying outliers or unusual patterns that do not conform to the learned expectations. Autoencoders provide an unsupervised, data-driven method for anomaly detection, particularly useful in scenarios where anomalous examples are scarce or difficult to label.

The last method reviewed in this paper, meta-learning, is intended to take advantage of the best aspects of two or more models in a combined model [50]. A meta-learning algorithm or meta-algorithm is a method to combine two or more algorithms to provide flexible systems that improve accuracy, performance, or other factors of interest [51]. Meta-algorithms are useful in any context that uses machine learning, but in particular they can be used to detect defects and anomalies in the data [52–54]. The purpose of this method is to combine the information provided by the primary methods to magnify their advantages while working together to reduce their disadvantages.

This research will utilize the three selected primary methods combined via an artificial neural network in a meta-algorithm to attempt to increase recall or precision while maintaining the other metric. Ideally, regression, ARIMA, and ANN together will increase both recall and precision,

which will lead to a better F_1 score. Section 3.3.5 contains details on the implementation of this technique.

2.3 Applications in Digital Twin Validation

This section will start by covering some specific examples of machine learning being used to detect defects and anomalies in sequential data in other domains unrelated to DTs. As the section progresses, it will get closer to machine learning being used to detect defects in DTs, but still not quite reach the main purpose of this paper, which is to detect defects in DT inputs in real time.

Mazumdar *et al.* utilized machine learning to determine defects in semiconductor chips by analyzing data collected from the chips sequentially as they left the manufacturing line [55]. The authors collected data as the chips were manufactured and tested individually and compared the new chips with data collected previously in the manufacturing process. An interesting detail about the study is that they used a meta-algorithm though they did not call it by that name. As part of the analysis, each chip was compared with previous data and if it had a coefficient of determination (R^2) greater than 0.7, the data were analyzed using regression and if less than 0.7 they used cluster analysis. They determined that each method was more effective under those circumstances. The technique is similar to this case study in that historical data was used to analyze new data, but the future was unknown.

In another use case, this time with image sensors, Tan *et al.* employ sequential data analysis methods to detect defects in the pixels of an image and then make corrections to the image [56]. They indicate that even a single defective pixel in an imaging sensor can have negative effects on surrounding pixels, so it is important not only to detect but also to correct anomalies. They use a sequential probability ratio test and then a minimum neighboring difference to identify bad pixels in an image. Then they suggest that you can use expectations from these methods to correct the pixel to a value that is reasonable and acceptable.

Wang *et al.* applies automated defect detection in a similar use case, where imaging is utilized to evaluate fabrics coming off a manufacturing process to detect defects and quality issues [57].

Due to the complexity and diversity of fabric patterns, the authors had to develop a sequential detection method in which images of fabrics were evaluated sequentially. The images were dynamically segmented based on the pattern into repetitive blocks so that the blocks could be compared against other blocks in real time. During the comparison step, the differences were flagged as defects, and then the type of defect could be categorized according to the shape of the defect.

Back in the domain of cybersecurity, Teng *et al.* proposed an approach to characterize user behavior and create a rulebase to store those patterns [58]. If a user's activity deviates significantly from the rulebase, anomalies are reported for security audits. The techniques used included a time-based inductive machine, which discovers patterns in sequential data, especially related to repetitive activities. These patterns can then be used to predict activity and, if there are differences between prediction and reality, identify anomalies. The method is constantly learning from the data stream and can dynamically adjust the model to account for which attributes of activity are important to the model.

Nizam *et al.* takes on the daunting task of analyzing large-scale dynamic data used by devices in the Internet of Things [59]. Data at this scale requires streamlined processes and low computational resources to make good decisions in real time. The paper uses a convolutional neural network and a long short-term memory (LSTM) based autoencoder to detect anomalies in time series data and deliver results in real time. The authors emphasize the importance of anomaly and rare event detection in Internet of Things environments and demonstrate that it is feasible to use machine learning to meet this goal efficiently.

The remaining papers to be analyzed will deal directly with digital twins. These papers are getting closer to the point and offer their own value, but differ from the efforts in this research in one or more key ways. At this point, it is important to mention that one DT's output could be another DT's input. The process of identifying defects in sequential inputs in real time could very well be handled by the data provider instead of the DT that accepts the data stream. However, the state-of-the-art way to verify and validate the output of a DT is to have a smarter approach to

performance analysis by comparing outputs to known expectations, referent data, or even the real system that the DT is modeling.

This means that a DT performing validation on its own data output has an advantage that a DT accepting that output as input may not have. The following works focus on the output of DTs rather than on the important step of validating inputs. As mentioned in Section 1.2, this paper will demonstrate that it is necessary and feasible to validate sequential inputs in real time, even without referent data.

Castellani *et al.* take the approach of using a DT to detect anomalies in real data [60]. If a DT is sufficiently realistic for the use case, it can generate ideal data against which a real system can be compared. The authors used a weekly supervised machine learning model that uses cluster centers and Siamese autoencoders to compare data that have very few labels. The advantage of using a DT in this fashion is that the DT can provide data that closely match well-known physical properties of the real system, giving ideal data or realistic samples of data. The authors demonstrate that this is a viable method with low false positive and false negative outcomes.

Using a DT in this way is an interesting use case. Usually modeling and simulation applications assume that the real system is the gold standard and strive to match it in every feasible way, but Castellani assumes that the DT is the standard. This can make sense in certain specific use cases where an ideal state is known, such as when modeling physical systems such as electrical or mechanical systems with established outcomes. If the idealized system is well understood, a DT that matches that ideal is a great candidate for comparison with a practical system.

In another interesting case of DT use, Xu *et al.* suggest a method to build DTs with live data provided by cyber systems, then using that model to detect anomalies [61]. The authors use a timed automaton machine as the digital twin of the cyber system, then use a generative adversarial network to detect anomalies, which uses an LSTM as the generator. The DT is continuously updated with new data flowing through the cyber system, which helps it better understand and identify anomalies.

The closest comparable approach to the concepts presented in this paper appears to be the work of Lugaresi *et al.* [18], which proposes the idea of validating digital twin outputs in real time by comparing them with a manufacturing process running in parallel to the model. By ensuring that the DT has the same inputs as the real-world system, a method that uses signal processing techniques is suggested to validate the DT during execution by comparing simulated outputs to the real system's outputs. The work is insightful when data are available, but does not cover the case where there is no referent data for comparison. In contrast, the research presented here expands on real-time validation by training models to detect defective inputs without referent data.

In summary, there is extensive literature reporting on the validation of digital twin outputs, defect detection, and sequential data analysis. In one case, these techniques were applied to DT outputs in a real-time environment to validate model performance, but this relied on a digital model running alongside a manufacturing process for comparison [18]. These techniques have not previously been applied to externally sourced sequential data inputs in a real-time DT environment with no comparison data. This work combines these topics to solve the real-world problem of analyzing DT inputs for defects in real time during a simulation run. It starts with a simple technique to demonstrate the viability of the idea and then suggests complex techniques that may be more robust for future research. The technique described in this paper is robust enough to detect defects in real time and is configurable to the sensitivity of the DT that it is protecting.

Chapter 3

Research Method

3.1 Setting

The case study setting includes data collection and analysis using the Python3 and R programming languages. Python was selected to perform the bulk of the data processing and data logging because it is flexible and ubiquitous in the machine learning world and is well supported by community-provided libraries [62]. R is used for defect detection techniques because of its vast collection of statistical computing libraries. The data are stored in a way that both Python and R have easy access to read and modify files.

Python's built-in pseudorandom number generator (RNG) is not necessarily repeatable across computers (operating system, processor architecture, etc.), so a repeatable random number generator was required. NumPy is a Python library with a repeatable RNG across computer architectures, but only for a fixed version of the package [63]. For this reason, all random numbers used in the Python code are generated by a file with a fixed NumPy version of 1.24.3. With the repeatable RNG in place, any user on any machine should be able to run the scripts and receive the same output. This ensures that this research is reproducible.

3.2 Data Collection

The case study focuses on rocket trajectories and attempts to identify a method to detect any defects in that trajectory over time. The validated RocketPy Python library provides an interface and the ability to randomize parameters to provide different types of trajectories for examination [64]. For each data set in the case study, 50 trajectories are generated randomly within the parameter constraints defined in Table 3.1. Other parameters available in RocketPy are fixed, such as starting elevation, atmospheric conditions, and modeled rocket type. After each trajectory is generated, the

three-dimensional position, velocity, and acceleration in 0.25 second increments is saved in a data structure for later analysis.

Table 3.1: Trajectory arguments, randomly selected between minimum and maximum values.

Trajectory Argument	Min Value	Max Value
Latitude (degrees)	-90	90
Longitude (degrees)	-180	180
Inclination (degrees)	45	85
Heading (degrees)	0	360
Burnout time (sec)	4	45

Figure 3.1 shows the X, Y, and altitude plots for Flight00001 generated by RocketPy. X and Y are coordinates expressed in meters from the rocket's origin at (0,0).

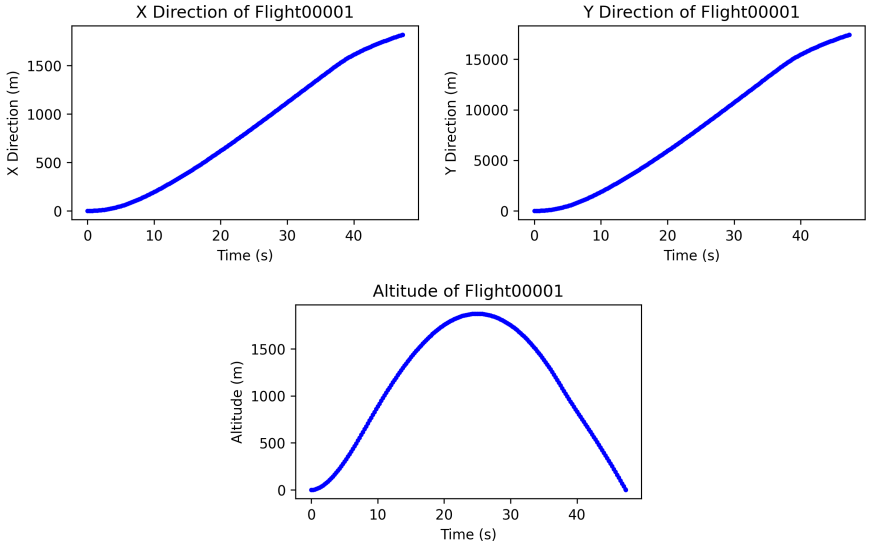

Figure 3.1: Flight00001 X, Y, and Altitude values over time.

Since RocketPy is a validated model of rocket trajectories, it is assumed that its data output is reliable. This means that RocketPy by itself does not produce trajectories for an interesting case

study with no defects to train a model to detect. It is necessary to introduce random defects to a few data points to test defect detection techniques. For each of the data sets in this study, of the 50 generated trajectories, 33 were randomly selected to contain defects. Of the trajectories identified to contain defects, one out of one-hundred (1%) trajectory updates was randomly selected to contain a defect. This resulted in 149 total defects introduced into the training data set. Of the trajectories with a defect, the minimum number of defects was 1, the maximum was 9, and the mean was 4.52.

Defects are introduced by centering the Gaussian (normal) distribution on the clean position data using the Box-Muller transform [65]. The Box-Muller transform requires two independent draws from a uniform distribution, U_1 and U_2, to generate a random draw from the normal distribution. Let (x_{ct}, y_{ct}, z_{ct}) be the clean rocket position (latitude, longitude, and altitude) at time t. The defective position, (x_t,y_t,z_t), is then calculated using Equation 3.1. For this case study, $\sigma = 0.75$ is chosen to provide somewhat uniform defect severity levels (low, medium, high) as described in Section 3.3. No defects were introduced in velocity or acceleration.

$$\begin{aligned} x_t &= x_{ct} + \sigma\sqrt{-2ln(U_1)}cos(2\pi U_2) \\ y_t &= y_{ct} + \sigma\sqrt{-2ln(U_3)}cos(2\pi U_4) \\ z_t &= z_{ct} + \sigma\sqrt{-2ln(U_5)}cos(2\pi U_6) \end{aligned} \tag{3.1}$$

Since rocket trajectories are so predictable at local scales, introducing too large a defect will make even the simplest defect detection technique look viable. To provide a challenge, the magnitudes of the defects are intentionally quite low and are not visible to the eye when inspecting the data visually (see the examples in Figure 4.1). Looking ahead to results, this leads to a lower recall rate, as some defects get lost in the noise, but this gives a better view into the performance of defect detection models.

3.3 Data Analysis

As a framework for evaluating defect detection techniques, it is useful to categorize defects into severity levels. The purpose of defect severity is to determine how much influence a defect may have on a DT. The impact on a DT will vary greatly based on the model's sensitivity to defects, so defect severity assignment should be tuned to a particular model and may have a level of subjectivity. Table 3.2 defines the severity levels.

Table 3.2: Defect severity levels used to evaluate defect detection techniques.

Severity	**Description**
Low	Defect causes minimal poor DT behavior
Medium	Defect causes some poor DT behavior
High	Defect causes serious DT behavior issues

This research defines the severity of a defect as the magnitude of deviation from expectation. A Kalman filter is a good option for assigning levels to defects because it can be tuned based on the sensitivity of the DT using its noise parameters [66]. For example, a DT of a radar signal might be very sensitive to defects compared to an air traffic controller display monitor, so it will have lower noise parameters. Other benefits of a Kalman filter include prediction and uncertainty calculation capabilities and its ability to adapt to new information while maintaining some memory of the past. These features mean that a Kalman filter would probably make a great defect detection technique on its own, but its value to this study is in the analysis of other defect detection techniques.

The Kalman filter process starts with an initial estimate of the state x_0 and assumed uncertainty P_0, where x_t is a vector of position, velocity, and acceleration and P_t is a 9x9 covariance matrix at time t. Then each state update z_t is fed into the filter to produce an updated x_t and P_t. The Kalman filter also generates a predicted state and uncertainty based on its current understanding, $\hat{x}_{t+1}$ and $\hat{P}_{t+1}$, respectively. The values used in each update and prediction step are in Table 3.3.

Table 3.3: Kalman filter parameters.

Attribute	**Type**	**Description**
x_t	Array length 9	State of trajectory at time t
P_t	Matrix 9x9	Covariance at time t
F	Matrix 9x9	State transition matrix
Q	Matrix 9x9	Process noise matrix
H	Matrix 3x9	Observation matrix
R	Matrix 3x3	Measurement covariance

At each step in a trajectory, the Kalman filter uncertainty (P_t, with σ_t as its diagonal) is used to compare the current, possibly defective, state x_t to the nondefective state x_{ct} to assign severity. All defects have at least a Low severity. A defect is a Medium severity if it is greater than one standard deviation away from x_{ct} and is a High severity if greater than two standard deviations as calculated by the Kalman filter (see Table 3.4).

Table 3.4: Definition of defect severity using Kalman filter uncertainty at time t, σ_t.

Defect Severity	**Definition**
Low	$\lvert x_t - x_{ct}\rvert <= \sigma_t$
Medium	$\sigma_t < \lvert x_t - x_{ct}\rvert <= 2\sigma_t$
High	$2\sigma_t < \lvert x_t - x_{ct}\rvert$

Table 3.5 shows the breakdown of the severity of the defects in all 50 trajectories in the training data, including the updates in the 17 trajectories that do not have defects. This data set is used to train the defect detection models using various parameter values.

Table 3.5: Training data total defects in 50 trajectories by severity.

Defect Severity	**Count**
None	21,369
Low	34
Med	78
High	37

Another set of 50 trajectories was generated using the same techniques described in Section 3.2, but with a different random seed. Table 3.6 displays the number of defects by severity for this validation data set. The validation data set is used to verify the defect detection models against independent data after training and to tune the parameters of the models.

Table 3.6: Validation data total defects in 50 trajectories by severity.

Defect Severity	**Count**
None	19,909
Low	33
Medium	79
High	26

Finally, Table 3.7 provides a breakdown of defects by severity for the test data set. This data is used as a final test of the defect detection models for viability and comparison between the models.

Table 3.7: Test data total defects in 50 trajectories by severity.

Defect Severity	Count
None	18,826
Low	35
Medium	72
High	25

3.3.1 Measuring Success

When comparing and tuning defect detection techniques, it is helpful to identify one or more metrics that are to be the benchmark of success of the technique. Many success measures strike a balance between true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Table 3.8 displays the common options used in machine learning with their definitions.

Table 3.8: Defect detection technique measures of success.

Measure	Definition	Description
Recall	$\frac{\text{TP}}{\text{TP+FN}}$	Proportion of defects that are detected
Precision	$\frac{\text{TP}}{\text{TP+FP}}$	Proportion of detections that have a defect
F_1 Measure	$\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision+recall}}$	Harmonic mean of recall and precision

Two common measures of success for categorization models are recall and precision [67]. Recall is the proportion of defects that are detected. Precision is the proportion of detections that actually have a defect, or the relevance of detected items. These measures are useful in defect detection because system owners are concerned with catching as many defects as possible (high recall) and minimizing resources wasted on false positives (high precision).

Other measures for binary classifiers, such as accuracy, specificity, and negative predictive value (NPV), are less useful. Accuracy is particularly problematic in situations similar to this data set, where there is a large imbalance between defects and good data [67]. For example, a baseline

model that declares that all data are good (there are no defects) has an accuracy of 99.3% in the training data because only 0.7% of the data are defective. This accuracy looks great superficially, but has exposed no defects and provided no information.

A better measure than precision or recall individually in such an inequality between defects and good data is their harmonic mean, the F_1 measure [67]. This metric weights the importance of recall and precision equally and ranges from zero to one. A value near or at zero implies that the detection technique is useless, and a value of one implies that the technique is perfect. This case study uses the F_1 measure to compare defect detection techniques.

A benefit of the F-measure when training a defect detection model is that the balance between recall and precision can be adjusted to the needs, appetite for risk, and available resources of the system owner. The generalized formula for the F_β-measure introduces β, a parameter that represents how many times more important recall is than precision. In this case study, we have used the F_1 measure that gives equal weight to recall and precision.

For a system owner who prioritizes catching as many defects as possible and has the resources to analyze false positives or a risk tolerance to allow them to be erroneously ignored by the digital twin, it may make sense to set β to be higher. On the other hand, the system owner may prefer higher precision because they do not have the resources to evaluate false positives, or they have a higher risk tolerance for defects slipping by. In that case, the model should be trained using a value of β less than one.

For any estimate of the F_1 measure, there is a confidence interval (CI) obtained using bootstrapping methods [68]. These confidence intervals are used to provide reasonable evidence that a defect detection technique is significantly better or worse than other techniques. A defect detection model is trained on the set of training data and then selected using the F_1 measure using a set of validation data. Then, the technique is applied to a set of test data to confirm the technique's capabilities.

In addition, three simple baseline models are provided for comparison with the regression technique. The first two baseline models provide measures for declaring that all data are a de-

fect ("Baseline - Always Defect") and that all data are good ("Baseline - Never Defect"). A third baseline model randomly declares that state updates are good or bad with 99.3% and 0.7% probability, respectively, which are the proportions of good and defective data. Comparing the regression technique with these baseline models provides evidence that the technique is useful.

Table 3.9 shows the results of the baseline models in the training data and Table 3.10 shows the success metrics. These defect detecting techniques demonstrate the importance of a good success measure, with some of the techniques having high accuracy, high negative prediction value, or even perfect specificity or recall. However, it is clear that none of these techniques are good tests, as they do not provide any real insight into defects in trajectory data. Using their values for the F_1 measure, all three models are at or near zero, which implies that they are useless tests.

Table 3.9: Baseline model results on training data.

Technique	**TrueNegative**	**TruePositive**	**FalsePositive**	**FalseNegative**
Baseline - Never Defect	21369	0	0	149
Baseline - Always Defect	0	149	21369	0
Baseline - Random	21216	1	153	148

Table 3.10: Baseline model measures of success.

Technique	**Recall**	**Precision**	$\mathbf{F_1}$
Baseline - Never Defect	0.000	N/A	0.014
Baseline - Always Defect	1.000	0.007	0.000
Baseline - Random	0.007	0.006	0.007

3.3.2 Second-order Polynomial Regression

In practice, the simplest defect detection techniques would likely start with an understanding of the underlying data. Using the example of a rocket trajectory that generally follows a parabolic

curve, at least locally, defect detection can use this to its advantage [69]. Therefore, this case study starts with the use of second-order polynomial regression to predict the next state update and compare it to the update that is provided. If the difference is great enough, it is marked as a defect.

Although the assumption is that a rocket trajectory is at least locally parabolic, it may not be perfectly parabolic on a large scale due to boost timing, atmospheric drag, and sources of perturbation such as wind conditions. The assumption is tested using a statistical kernel to weigh a window of trajectory updates to predict the next state so that only the most recent updates are included in the model [70]. This kernel is a window of time in the past that is used to predict the next state and is tuned to form the best model.

The second-order polynomial regression formula is shown in Equation 3.2, where $\hat{X}$ is the predicted state of the trajectory based from the regression on time for the last T seconds of updates, $t_{[0,T]}$. Only trajectory updates that had available data at least as large as the window are evaluated for defects. Any set of data smaller than this is not evaluated, as it does not have enough data.

$$\hat{X} = \beta_0 + \beta_1 t_{[0,T]} + \beta_2 t_{[0,T]}^2 \tag{3.2}$$

Using the regression model, the estimated trajectory is propagated forward in time for any Δt to $\hat{X}_{\Delta t}$. A special type of confidence interval called a prediction interval is formed around $\hat{X}_{\Delta t}$ as shown in Equation 3.3 [71]. This prediction interval is an estimate of an interval in which a future observation will fall with a specified confidence $(1 - \alpha)$ applied to a t-statistic and using the predicted value's standard error (SE). In the case study, this interval is fixed at 99% confidence with a multiplier (m) to adjust the lower and upper bounds to tune the model.

$$\hat{X}_{\Delta t} \pm m t_{\alpha/2} SE \tag{3.3}$$

Let the prediction interval with multiplier minimum and maximum be notated [LB, UB] for the lower bound (LB) and the upper bound (UB). Then, when the trajectory update is processed and

compared to the prediction, it is flagged as a defect if it falls outside of these prediction interval limits (see Figures 3.2 and 3.3).

The model is executed against the training data tuning several combinations of values of the time window T and the interval multiplier m. The best model is selected and is evaluated against the independent test data set.

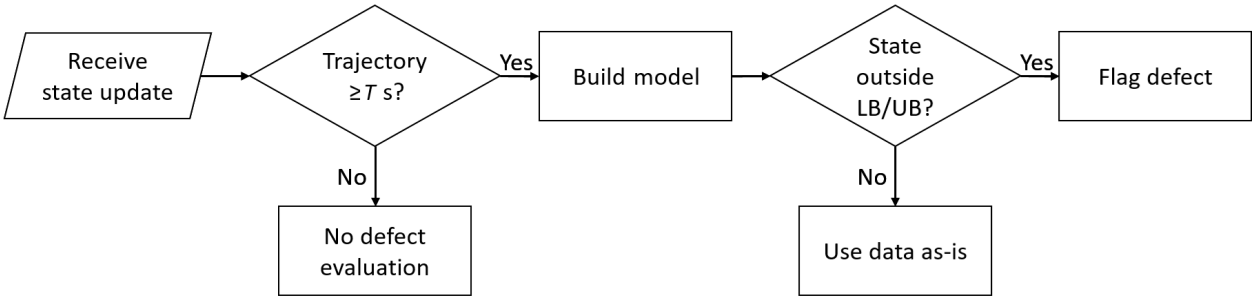

Figure 3.2: Defect detection workflow.

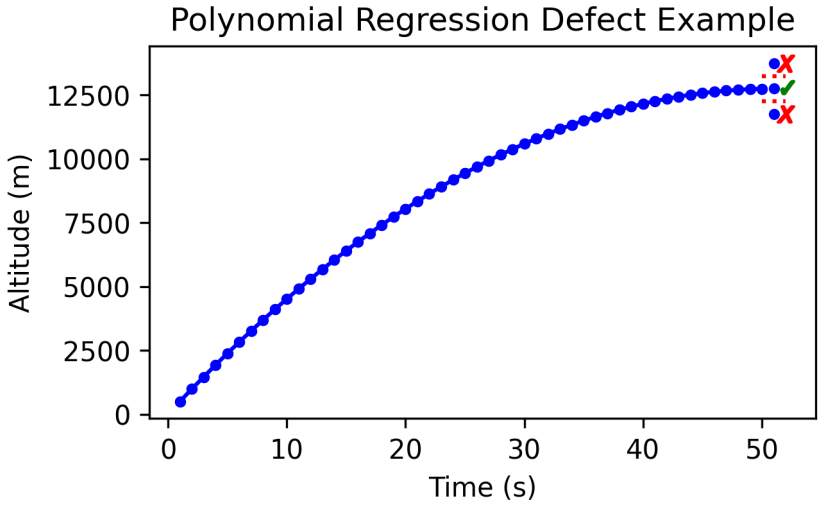

Figure 3.3: Example trajectory with LB and UB (dashed red line). If above or below these limits, the next trajectory update is noted as a defect.

As the multiplier increases, the range between LB and UB also increases. It is then expected that increasing the multiplier will reduce the model's ability to capture defects (lower recall), while also reducing false positives (higher precision). The balance between these goals is found in max-

imizing the F_1 measure, as described in 3.3.1. The list of parameters that are tuned for this model is in Table 3.11.

Table 3.11: Parameters to be tuned in regression model.

Parameter	**Symbol**	**Description**
Lag Time	T	Time in seconds of historical data to use
Interval Multiplier	m	A multiplier to adjust the lower and upper bounds.

Some advantages to this technique include its simplicity and ability to build a model in real time with no pre-execution training. This adaptability presents the opportunity to implement the technique without the requirement of an a priori database and without pre-existing knowledge of the data other than the assumption that it is locally parabolic in nature. One disadvantage of this technique is that it is not robust enough for all types of sequential data, so other models might be necessary for different situations. Another downside of the technique is that its decisions are binary, so the severity of the difference between the prediction and the actual trajectory is only used to determine whether an update is a defect or not without assigning a severity level. However, it is anticipated that this technique will be more likely to catch the most severe defects in the medium and high severity categories rather than low severity defects, which is beneficial to DTs.

3.3.3 ARIMA

After fitting a model that is expected to be very predictive for certain data shapes, the next step is to have a more general approach that could be used in a wider diversity of data. The autoregressive integrated moving average (ARIMA) technique shows promise, as it is capable of modeling other data shapes and can adapt to changing conditions more fluidly [32]. ARIMA models are appreciated in time series analysis for their flexibility and ability to model various data behaviors. This allows ARIMA to adapt to a wide range of time series patterns, including non-stationary data that exhibit trends or seasonal behaviors. Although specialized models excel

in specific contexts, integrating ARIMA into this toolkit provides a more adaptable approach. It broadens the scope of data types that can be effectively modeled and supports ongoing adjustments necessary for accurate forecasting in complex situations.

The regression model assumed that a rocket trajectory is at least locally parabolic, but it may not be perfectly parabolic due to boost timing, atmospheric drag, and sources of perturbation such as wind conditions. However, similar to the regression model, ARIMA uses a statistical kernel to weigh a window of trajectory updates to predict the next state so that only the most recent updates are included [70]. This kernel is a window of time (T) in the past used to predict the next state and is tuned to form the best model. Only trajectory updates that had available data at least as large as the window are evaluated for defects. Any set of data smaller than this is not evaluated, as it does not have enough data.

ARIMA models have three primary components: autoregression (AR), differencing applied to the time series to make it stationary (I), and moving average (MA). The autoregressive component is controlled by the parameter p and represents the relationship between a current value and its past values. The integrated component (I) represents the differencing needed to make the time series stationary and is controlled by the parameter d. The moving average component is controlled by q and represents the relationship between a current value and past forecast errors.

Autoregression and moving average can each individually be powerful forecasters, but require the data to be stationary [72]. Data are considered stationary if their mean and variance remain constant over time [73]. Without this property, autoregression, in particular, can make a poor model and may not be very predictive. The differencing component of ARIMA corrects issues with stationarity so that the other components can function better.

Using the ARIMA model, the estimated trajectory is propagated forward in time for any Δt to $\hat{X}_{\Delta t}$. A special type of confidence interval called a prediction interval is formed around $\hat{X}_{\Delta t}$ as shown in Equation 3.4 [71]. This prediction interval is an estimate of an interval in which a future observation will fall with a specified confidence $(1 - \alpha)$ applied to a t-statistic and using

the predicted value's standard error (SE). In the case study, this interval is fixed at 99% confidence with a multiplier (m) to adjust the lower and upper bounds to tune the model.

$$\hat{X}_{\Delta t} \pm m t_{\alpha/2} SE \tag{3.4}$$

Let the prediction interval with multiplier minimum and maximum be notated [LB, UB] for the lower bound (LB) and the upper bound (UB). Then, when the trajectory update is processed and compared to the prediction, it is flagged as a defect if it falls outside of these prediction interval limits (see Figure 3.4).

The model is executed against the training data tuning several combinations of the values of the ARIMA parameters (p, d, q), the time window T, and the interval multiplier m. The best model is selected and evaluated against the independent test data set.

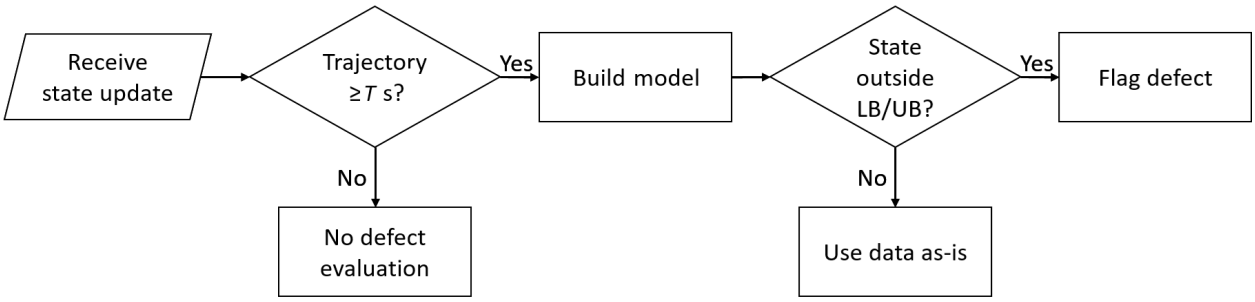

Figure 3.4: Defect detection workflow.

As the multiplier increases, the range between LB and UB also increases. It is then expected that increasing the multiplier will reduce the model's ability to capture defects (lower recall), while also reducing false positives (higher precision). The balance between these goals is found in maximizing the measure F_1, as described in 3.3.1. The list of parameters that are tuned for this model is in Table 3.12.

Table 3.12: Parameters to be tuned in ARIMA model.

Parameter	**Symbol**	**Description**
Autoregressive (AR)	p	Order of the autoregressive (AR) component
Integrated (I)	d	Degree of differencing applied to make data stationary.
Moving Average (MA)	q	Order of the moving average (MA) component.
Lag Time	T	Time in seconds of historical data to use
Interval Multiplier	m	A multiplier to adjust the lower and upper bounds.

Similarly to regression, advantages to ARIMA include its simplicity and ability to build a model in real time with no pre-execution training. This adaptability presents the opportunity to implement the technique without the requirement of an a priori database and without pre-existing knowledge of the data. ARIMA is also not very computationally intense, so it holds promise for building models and identifying defects in real-time DTs. Like with regression, one downside of the technique is that its decisions are binary, so the severity of the difference between the prediction and the actual trajectory is only used to determine whether an update is a defect or not without assigning a severity level. Another disadvantage is that, while ARIMA is still interpretable and somewhat simple, it does not have the simplicity of a regression model. It is expected that this technique will do about as well as regression, since at worst it will tune to a model that closely represents a polynomial regression model.

3.3.4 Artificial Neural Network

The techniques discussed so far make at least some assumptions about the data used to fit the models. However, artificial neural networks (ANN) make few assumptions about the data and will try their best to find connections and trends regardless [33]. This makes ANN powerful tools and useful for an even more broad set of data types than ARIMA, but it may sacrifice some predictability or precision for this generality. Due to the simple nature of the data, a single-layer network is sufficient, although there are other parameters that must be tuned [34].

An example of how an ANN model works is shown in Figure 3.5, where $\hat{X}$ is the predicted three-dimensional state of the trajectory based on data from the last T seconds of updates, $t_{[0,T]}$. Only trajectory updates that had available data at least as large as the window are evaluated for defects. Any set of data smaller than this is not evaluated, as it does not have enough data. This ANN example illustrates how the data input flows through the network nodes to make a decision using a set of weights. Each input and node has the opportunity to affect subsequent nodes.

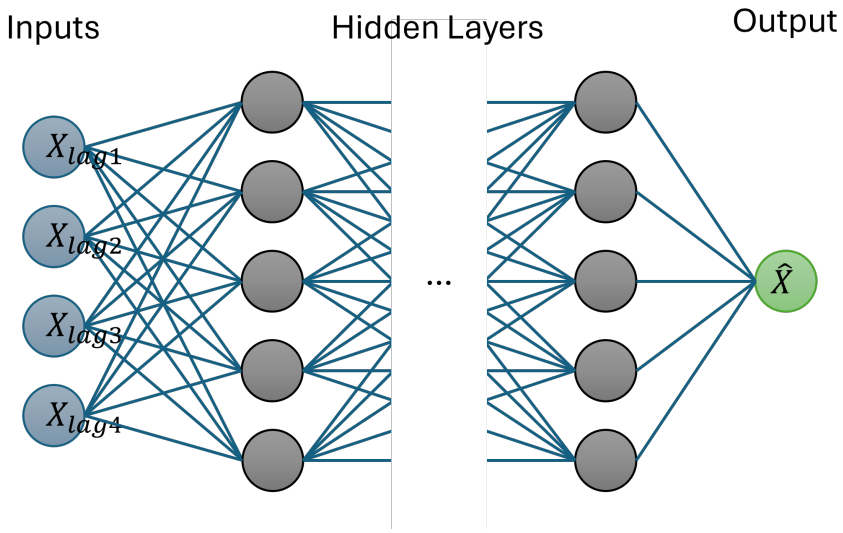

Figure 3.5: Example artificial neural network.

In an artificial neural network model, the nodes are the fundamental processing units that receive input data, perform computations, and pass information forward. These nodes are organized into layers: the input layer, one or more hidden layers, and the output layer. Hidden layers are intermediate layers located between the input and output layers and play a critical role in enabling the network to learn complex patterns. The number of nodes in the hidden layer is one of the parameters that was optimized by training and validation. Each node in a hidden layer receives weighted inputs from nodes in the previous layer, sums these inputs, and then applies an activation function [74]. This process allows the network to model complex relationships within the data. During training, the network adjusts the weights associated with the connections between nodes

to improve its performance, which helps the hidden layers to find important patterns and features, helping the network to make accurate predictions or categorize things correctly.

The other ANN parameter that was optimized in training is weight decay. Weight decay is a technique used in the training of artificial neural networks to prevent overfitting and improve generalization [75]. It works by adding a penalty term to the loss function that is proportional to the magnitude of the network's weights. During optimization, this penalty encourages the model to keep its weights small by effectively shrinking them toward zero at each update step. As a result, weight decay discourages complex models that fit the training data too closely, promoting simpler models that are more likely to perform well on other data sets by reducing overfitting.

The ANN model also differs from the previous two in that it was trained on the clean, unmodified state data in the training set. The reason for this is that the ANN method will be a supervised model that needs to see data free from defects to provide an accurate model. After being trained, the model is applied to the validation data set with full data defects so that it provides a clear picture of how well the model works in identifying those defects. The validation data set is also where the model parameters are determined, since it is independent of the training data and contains defects. Training on one set and validating on another will also reduce the risk of overfitting [76].

Using the ANN model, the estimated trajectory is propagated forward in time for any Δt to $\hat{X}_{\Delta t}$. Without a closed-form solution to create a confidence interval using this method, bootstrapping was utilized to create the interval and, like the other methods, a multiplier (m) is applied to adjust the lower and upper bounds to tune the model. This method turned out to be computationally intense as various models with starting weights were applied to predict $\hat{X}_{\Delta t}$, then bootstrapping was applied to those data to obtain the interval.

Let the prediction interval with multiplier minimum and maximum be notated [LB, UB] for the lower bound (LB) and the upper bound (UB). Then, when the trajectory update is processed and compared to the prediction, it is flagged as a defect if it falls outside of these prediction interval limits (see Figure 3.6).

The model is executed against the training data tuning several combinations of values of the time window T, the interval multiplier m, and the decay and hidden size of the model parameters. The best model is selected using the validation data set and is evaluated against the independent test data set.

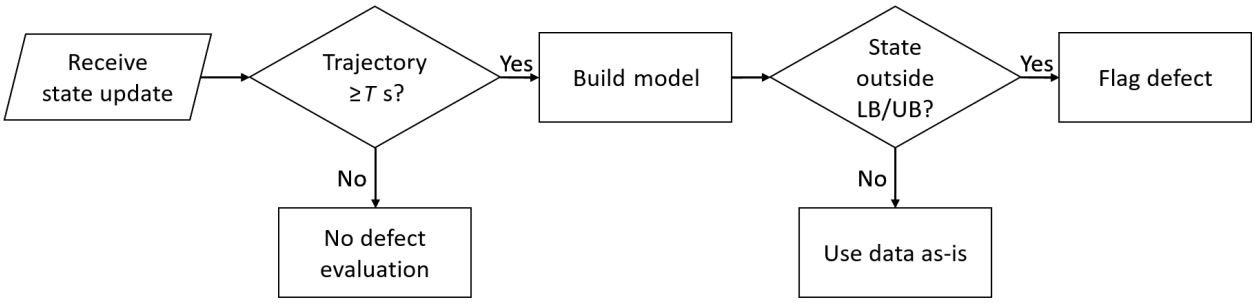

Figure 3.6: Defect detection workflow.

As the multiplier increases, the range between LB and UB also increases. It is then expected that increasing the multiplier will reduce the model's ability to capture defects (lower recall), while also reducing false positives (higher precision). The balance between these goals will be found by maximizing the F_1 measure, as described in 3.3.1. The list of parameters that will be tuned for this model is in Table 3.13.

Table 3.13: Parameters to be tuned in ANN model.

Parameter	**Symbol**	**Description**
Lag Time	T	Time in seconds of historical data to use
Interval Multiplier	m	A multiplier to adjust the lower and upper bounds.
Hidden Size		Number of nodes in the hidden layer
Decay		Weight decay

Some advantages of this technique include its flexibility and generalization, especially in comparison to previous techniques. ANNs are great for modeling complex relationships and may adapt

to patterns, even when not explicitly designed to the pattern. ANN also learns relevant features in the data automatically and will give them higher weights compared to features that do not affect the outcome. This complexity comes at the cost of the model being a sort of "black box" where a user will have difficulty interpreting the results. This lack of understanding of the inner workings of the model means that users largely accept the outcome and cannot analyze issues with the model. In addition, ANN requires more data and computational resources to train and execute the model. This can be partially solved in some use cases by training the model as a pre-processing step and then applying it to data live during a DT's execution.

3.3.5 Meta-Algorithm

The final technique in this case study involves combining the outputs of the other models in an attempt to maximize the advantages of each method while minimizing the disadvantages [50]. A meta-learning algorithm or meta-algorithm is a method to combine two or more algorithms to provide flexible systems that improve accuracy, performance, or other factors of interest [51]. The intent of this technique is to amplify the best of each model and reduce the impact of where they go wrong. For this case study, an artificial neural network (ANN) with one hidden layer was used as the meta-algorithm with the three primary models as input.

An example of how this meta-algorithm works is shown in Figure 3.7. The inputs Y_{reg}, Y_{ARIMA}, and Y_{ANN} are the defect determinations from the regression, ARIMA, and ANN techniques, respectively, for each trajectory update. The input T_p is a time-based component that is explained in the next paragraph. These four inputs are passed through an ANN to determine the output $\hat{Y}$, which is a binary decision that predicts whether that trajectory update has a defect based on the three input models and possibly the time component.

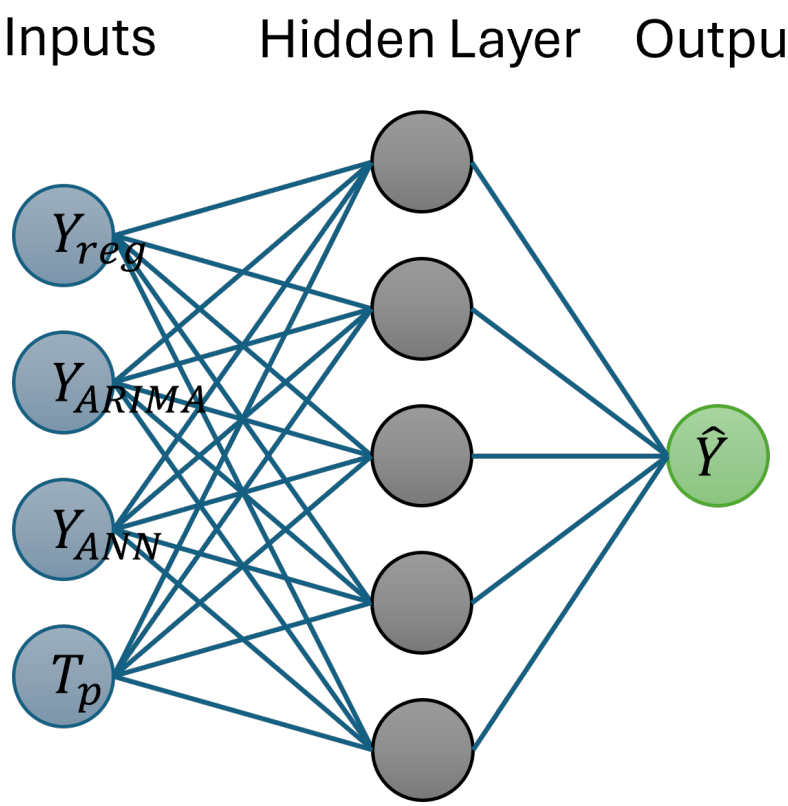

Figure 3.7: Example meta-algorithm using an artificial neural network.

In addition to the hidden layer size and decay parameters that were trained for the previous ANN technique, an additional time-based parameter (T_p) is introduced. This time component is a proportion between zero and one, inclusive, of the time that has passed in a trajectory compared to its total flight time. For example, investigating an update to training data Flight00003 at time 12.75 seconds, which has a total flight time of 177.3 seconds, gives a time component value of $T_p = 12.75/177.3 = 0.072$, indicating that this update is at 7.2% of the time into the flight. When selecting the best model based on values for the hidden layer size and decay parameters, the time component will also be selected to be included by training and validating the models with and without the component.

The benefit of this parameter is to allow the model to give more or less weight to the three primary techniques depending on which stage of flight the trajectory update comes in. Perhaps ARIMA does really well in the early stages of flight, but poorly in the middle where regression

excels. The meta-algorithm can use the time component to train the model to weigh each primary technique differently at different stages of flight. It should be mentioned that, in real-world use cases, a total or estimated total flight time may not be available, in which case this technique would rely entirely on the primary models without the time component. Some real-world systems could estimate total flight time, such as a radar DT that can estimate the time to impact of a trajectory.

The meta-algorithm model is quite unlike the previous three techniques because it does not use much of the data included with each update. With the exception of the time component, nothing about the trajectory itself (position, velocity, etc.) is included in the model. It relies almost entirely on the results of the three primary models: regression, ARIMA, and ANN.

The model is executed against the training data, tuning several combinations of values of the model parameters decay and hidden size. This process is repeated twice, once without the time component and once with the component. The best model is selected using the F_1 value against the validation data set and is evaluated against the independent test data set for comparison to the other techniques. Training on one set and validating on another will also reduce the risk of overfitting the model [76].

Table 3.14: Parameters to be tuned in meta-algorithm model.

Parameter	**Description**
Hidden Size	Number of neurons in a hidden layer
Decay	Weight decay
Time Component	Include or exclude flight time component

An advantage of this technique includes the use of the strengths of each primary model, leading to potentially higher overall accuracy and robustness. This collective approach can reduce overfitting and improve generalization to new data. Additionally, the meta-algorithm can learn to weigh or combine the outputs of the three models dynamically, adapting to varying data contexts, such as flight stage of the trajectory. However, there are also notable disadvantages. The

increased complexity can lead to higher computational costs and longer training times, making its use more resource intensive. Also, if one or more of the primary models are performing poorly in a certain context, these issues can propagate or amplify through the meta-algorithm, potentially degrading overall performance. Justification for using a meta-algorithm must include analysis into the computational resources and real-time performance of the technique.

3.4 Research Questions

The purpose of this study is to provide evidence that detecting defects in sequential data streams before they are processed by a digital twin is possible, useful, and critically important. The following sections define the success for each of the research questions provided in Section 1.3.

3.4.1 RQ1: Detecting Defects in Sequential Inputs

Research Question 1: Can sequential input data with defects be detected in real time using second-order polynomial regression, ARIMA, and artificial neural networks?

This research question will have a positive outcome if the three primary techniques in Section 3.3 are able to demonstrate that they provide information above and beyond the baseline models. Since the results of the baseline model are not impressive (see Table 3.10), this case study will set a higher bar for success. Although the definition of a "good" F_1 measure is subjective and heavily dependent on the use case, some sources consider a value above 0.5 to be useful [77–79]. So, this research question will be successful if $F_1 > 0.5$ for all three primary defect detection techniques.

3.4.2 RQ2: Using a Meta-Algorithm

Research Question 2: Do these defect detection techniques have better performance working together in a meta-algorithm?

The criteria for a successful meta-algorithm will depend on the results from the three primary defect detection techniques that are used an inputs. As designed in Section 3.3.5, the meta-algorithm needs to be at least as good as the three primary techniques and ideally would exceed

their performance. This research question will be successful if the meta-algorithm F_1 value exceeds that of the three primary techniques individually.

3.4.3 RQ3: Practical Use in Real-world Applications

Research Question 3: Are these techniques applicable to practitioners in the real world?

Although this research intends to convince the reader of the importance of applying defect detection to sequential data input by proving that it is possible in RQ1 and RQ2, a practitioner will need to carefully evaluate whether the effects of defects are detrimental to their DT. Chapter 1 provides numerous reasons to implement defect detection and outlines several ways in which a DT can fail if bad data is allowed to drive results. The application to researchers and practitioners is extensively discussed in Section 5.2. The success of the techniques discussed in this case study along with the discussion will demonstrate the applicability to real-world use cases.

Chapter 4

Presentation of Results

Section 3.2 provides the data collection strategy for this study. Following that description, the defect detection models were trained using a training data set, the model parameters were optimized on a validation set, and the results were compared on a test set. Each update may or may not be defective. The Kalman filter strategy categorizes each defect as low, medium, or high per Section 3.2 or "none" if no defect is present. The three baseline models were applied to the data to establish minimum performance expectations as described in Section 3.3.1. The results of the baseline models are given in Tables 3.9 and 3.10.

4.1 Second-order Polynomial Regression

Following the research design in Section 3.3.2, a kernel-smoothed second-order polynomial regression model is applied to each trajectory update that has enough historical data (at least T seconds of updates). Several time windows (T) and multipliers (m) were tested to determine the best approach. The results of this defect detection analysis are given in Table 4.1.

Table 4.1 displays a heat map with lower F_1 values in red and higher in green. It gives a quick look at how tuning the model focused on the best parameters with an interval multiplier of 20 and a past update window of 6, representing 1.5 seconds.

Table 4.1: 2nd-order polynomial regression results.

	Past Updates Window								
Multiplier	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	0.570	0.575	0.403	0.299	0.244	0.205	0.177	0.147	0.126
2	0.599	0.717	0.684	0.678	0.672	0.645	0.633	0.615	0.602
5	0.602	0.772	0.767	0.745	0.731	0.721	0.707	0.691	0.683
10	0.562	0.757	0.771	0.743	0.739	0.735	0.725	0.717	0.714
20	0.470	0.766	**0.779**	0.762	0.739	0.734	0.718	0.734	0.694
50	0.308	0.706	0.747	0.748	0.698	0.681	0.628	0.545	0.512
100	0.246	0.571	0.608	0.569	0.505	0.490	0.419	0.413	0.370
150	0.180	0.462	0.490	0.482	0.419	0.385	0.370	0.333	0.305

Figure 4.1 illustrates one example each of the four potential outcomes: true positive (top left), true negative (top right), false positive (bottom left), and false negative (bottom right). Since the scales of defects are often small compared to the analysis window, the expanded portion of the images show the received update compared to the expected trajectory (blue line) and confidence bounds (red dotted lines).

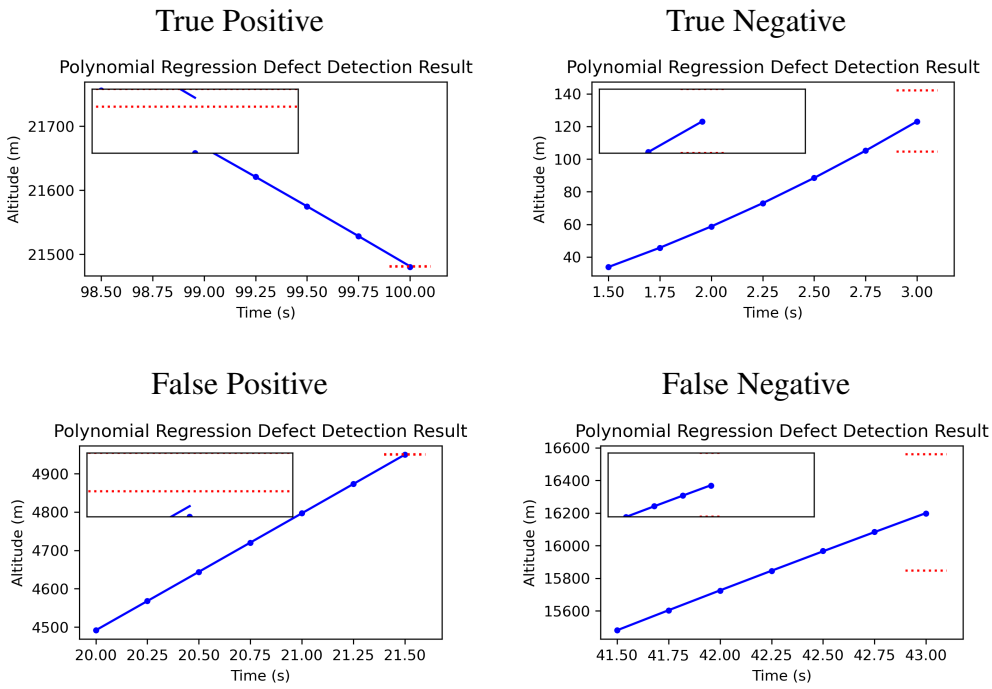

Figure 4.1: Polynomial regression defect detection result examples.

The success metrics defined in Section 3.3.1 are calculated and displayed in Table 4.2 for the regression model using a multiplier of 20 and a window of 6 on the validation data.

Table 4.2: 2nd-order polynomial regression success metrics.

Metric	**Value**	**LB**	**UB**
Recall	0.717	0.633	0.789
Precision	0.762	0.677	0.830
F_1	0.739	0.680	0.800

This method for detecting defects results in a recall of 71.7% (95% CI [63.3, 78.9]) and a precision of 76.2% (95% CI [67.7, 83.0]). This means that approximately 72% defects were detected and that roughly 76% of the detections were true positives, while the remainder were false positives. The lower precision may or may not be acceptable depending on the use case. After

a detection occurs, if the state update is thrown out and this has minimal effect on the DT, lower precision may be allowable. On the other hand, if an analyst manually reviews each detection, false positives may be very costly to adjudicate, and a decision will need to be made to balance precision and other measures.

Table 4.3: Defect severity breakdown for second-order polynomial regression.

Defect Severity	True Positive	False Negative	Recall	LB	UB
Low	16	17	0.485	0.308	0.665
Medium	60	19	0.759	0.650	0.849
High	23	3	0.885	0.728	0.968

Table 4.3 provides a performance breakdown by defect severity level for the regression technique on validation data. It is not surprising to observe that the recall for low-severity defects is significantly less than for medium- and high-severity defects. It is also not surprising to see that High severity had the highest recall rate, although it was not significantly better than Medium when comparing the bounds.

4.1.1 Test Data

After demonstrating that second-order polynomial regression using an interval multiplier of 20 and a window of 6 is a viable technique to detect defects in a sequential data feed, it was applied now to the set of test data. Reference Tables 4.4 and 4.5 for the test data defect detection results. This method results in a recall of 77.3% (95% CI [69.0, 83.9]) and a precision of 86.4% (95% CI [78.6, 91.8]). This means that approximately 77% defects were detected and that roughly 86% of all detections were true positives, with the remainder being false positives. The F_1 measure is 0.816 (95% CI [0.765, 0.870]). These results are comparable to the metrics observed in the validation data.

Table 4.4: Confusion matrix for second-order polynomial regression applied against the test data.

		Predicted		
		Positive	Negative	Total
Actual	Defect	102	30	132
	Good	16	18,810	18,826
	Total	118	18,840	18,958

Table 4.5: Summary of measures of success for second-order polynomial regression applied against the test data.

Success Measure	**Value**	**LB**	**UB**
Recall	0.773	0.690	0.839
Precision	0.864	0.786	0.918
F_1	0.816	0.765	0.870

Table 4.6 provides a performance breakdown by defect severity level for the second-order polynomial regression technique applied to test data. Once again, low-severity defects were slightly less likely to be detected than medium and high-severity defects.

Table 4.6: Test data defect severity breakdown for second-order polynomial regression.

Defect Severity	**True Positive**	**False Negative**	**Recall**	**LB**	**UB**
Low	21	14	0.600	0.421	0.761
Medium	61	11	0.847	0.743	0.921
High	20	5	0.800	0.593	0.932

When comparing the results of the test data with the metrics of the validation data, the second-order regression defect detection technique is at least as effective in the independent test data as it

was in the validation data. This demonstrates the viability of this technique in detecting defects in sequential data for trajectories, which may be input for a DT.

4.2 ARIMA

Following the research design in Section 3.3.3, an ARIMA model is applied to each trajectory update that has enough historical data (at least T seconds of updates). Several order values (p, d, q), time windows (T), and multipliers (m) were tested to determine the best approach. Of the top results for ARIMA(p,d,q), the best F_1 values were ARIMA(2,2,0) at 0.663 and ARIMA(2,1,0) at 0.629. Due to the higher F_1 value, the results of the ARIMA(2,2,0) defect detection model are given in Table 4.7.

Table 4.7 displays a heat map with lower F_1 values in red and higher in green. It gives a quick look at how tuning the ARIMA(2,2,0) model focused on the best parameters with an interval multiplier of 600 and a past update window of 8, representing 2.0 seconds.

Table 4.7: ARIMA(2,2,0) results.

	Past Updates Window			
Multiplier	**7**	**8**	**9**	**10**
100	0.551	0.634	0.610	0.604
150	0.588	0.656	0.613	0.607
200	0.600	0.653	0.616	0.602
250	0.603	0.643	0.597	0.587
300	0.611	0.647	0.604	0.590
350	0.624	0.656	0.606	0.584
400	0.637	0.657	0.604	0.583
450	0.646	0.651	0.603	0.582
500	0.638	0.655	0.598	0.582
600	0.648	**0.663**	0.599	0.588
700	0.644	0.659	0.599	0.578
800	0.642	0.661	0.597	0.574
900	0.638	0.661	0.593	0.575
1000	0.640	0.659	0.582	0.571
1200	0.640	0.650	0.573	0.570
1400	0.639	0.654	0.571	0.552
1600	0.650	0.630	0.555	0.556

Figure 4.2 illustrates one example each of the four potential outcomes: true positive (top left), true negative (top right), false positive (bottom left), and false negative (bottom right). Since the scales of defects are often small compared to the analysis window, the expanded portion of the images show the received update compared to the expected trajectory (blue line) and confidence bounds (red dotted lines).

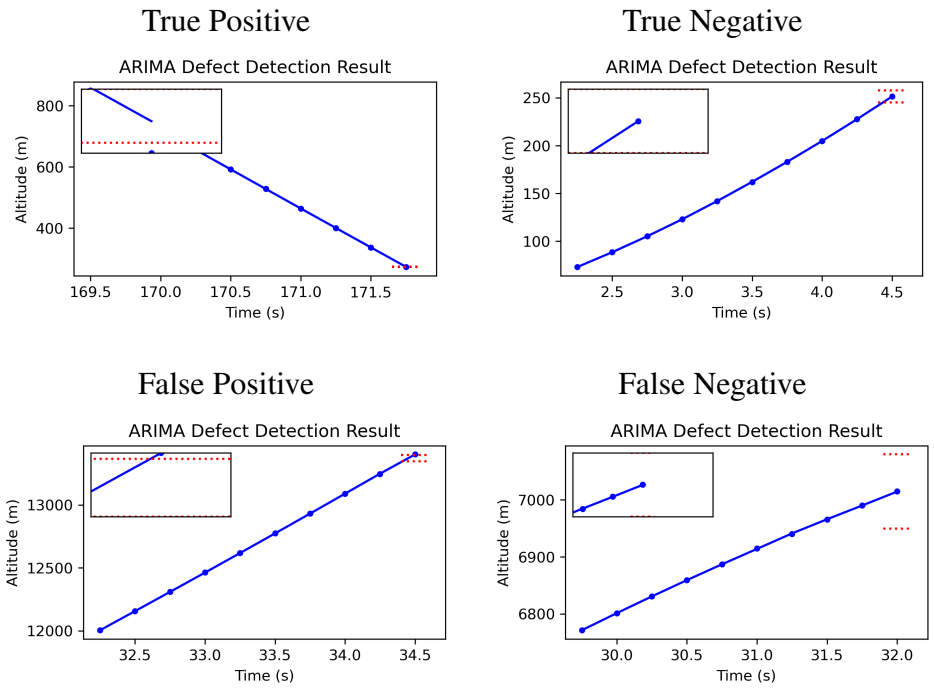

Figure 4.2: ARIMA(2,2,0) defect detection result examples.

The success metrics defined in Section 3.3.1 are calculated and displayed in Table 4.8 for the ARIMA(2,2,0) model using a multiplier of 600 and a window of 8 on the validation data.

Table 4.8: ARIMA(2,2,0) success metrics.

Metric	**Value**	**LB**	**UB**
Recall	0.797	0.718	0.859
Precision	0.512	0.443	0.580
F_1	0.623	0.566	0.680

This method for detecting defects results in a recall of 79.7% (95% CI [71.8, 85.9]) and a precision of 51.2% (95% CI [44.3, 58.0]). This means that approximately 80% defects were detected and that roughly 51% of the detections were true positives, with the remainder being false positives. The lower precision may or may not be acceptable depending on the use case. After

a detection occurs, if the state update is thrown out and this has minimal effect on the DT, lower precision may be allowable. On the other hand, if an analyst manually reviews each detection, false positives may be very costly to adjudicate, and a decision will need to be made to balance precision and other measures.

Table 4.9: Defect severity breakdown for ARIMA(2,2,0).

Defect Severity	True Positive	False Negative	Recall	LB	UB
Low	21	12	0.636	0.472	0.800
Medium	66	13	0.835	0.754	0.917
High	23	3	0.885	0.710	0.960

Table 4.9 provides a performance breakdown by defect severity level for the ARIMA technique on validation data. It is not surprising to observe that the recall for low-severity defects is significantly less than for medium- and high-severity defects. It is also not surprising to see that High severity had the highest recall rate, although it was not significantly better than Medium when comparing the bounds.

4.2.1 Test Data

After demonstrating that ARIMA using an interval multiplier of 600 and a window of 8 is a viable technique to detect defects in a sequential data feed, it was now applied to the set of test data. Reference Tables 4.10 and 4.11 for the test data defect detection results. This method results in a recall of 80.3% (95% CI [72.3, 86.5]) and a precision of 56.7% (95% CI [49.3, 63.8]). This means that approximately 80% defects were detected and that roughly 57% of all detections were true positives, with the remainder being false positives. The F_1 measure is 0.665 (95% CI [0.605, 0.725]). These results are comparable to the metrics observed in the validation data.

Table 4.10: Confusion matrix for ARIMA(2,2,0) applied against the test data.

		Predicted		
		Positive	Negative	Total
Actual	Defect	106	26	132
	Good	81	18,745	18,826
	Total	187	18,771	18,958

Table 4.11: Summary of measures of success for ARIMA(2,2,0) applied against the test data.

Success Measure	**Value**	**LB**	**UB**
Recall	0.803	0.723	0.865
Precision	0.567	0.493	0.638
F_1	0.665	0.605	0.725

Table 4.12 provides a performance breakdown by defect severity level for the ARIMA technique applied to the test data. Once again, low-severity defects were slightly less likely to be detected than medium and high-severity defects.

Table 4.12: Test data defect severity breakdown for ARIMA(2,2,0).

Defect Severity	**True Positive**	**False Negative**	**Recall**	**LB**	**UB**
Low	22	13	0.600	0.421	0.761
Medium	63	9	0.847	0.743	0.921
High	21	4	0.800	0.593	0.932

When comparing the results of the test data with the metrics of the validation data, the ARIMA defect detection technique is at least as effective in the independent test data as it was in the

validation data. This demonstrates the viability of this technique in detecting defects in sequential data for trajectories.

4.3 Artificial Neural Network

Following the research design in Section 3.3.4, an ANN model is trained to each trajectory update that has enough historical data (at least T seconds of updates) using the training data and then the model is applied to the validation data set. Several decays, hidden sizes, lags (T), and multipliers (m) were tested to determine the best approach (see Table 4.13). Of the top results for ANN, the best F_1 value was 0.592 for ANN with decay 0.0001 and using $T = 1$ second of historical data. Due to the higher F_1 value, the results of the ANN with the decay 0.0001 defect detection model are given in Table 4.14.

Table 4.13: ANN results by decay value.

Decay	**Best $\mathbf{F_1}$**
0	0.559
0.0001	**0.592**
0.001	0.556
0.01	0.577
0.1	0.574
1	0.562

Table 4.14 displays a heat map with lower F_1 values in red and higher in green. It gives a quick look at how tuning the ANN with decay 0.0001 model focused on the best parameters with an interval multiplier of 400 and a hidden size of 14.

Table 4.14: ANN results with decay 0.0001 and T=0.

	Hidden Size				
Multiplier	**12**	**13**	**14**	**15**	**16**
150	0.415	0.366	0.430	0.413	0.382
200	0.435	0.373	0.466	0.443	0.369
250	0.472	0.381	0.513	0.476	0.357
300	0.479	0.389	0.556	0.505	0.354
350	0.486	0.390	0.567	0.509	0.362
400	0.479	0.417	**0.592**	0.522	0.318
450	0.482	0.436	0.570	0.534	0.282
500	0.476	0.438	0.564	0.540	0.246
550	0.486	0.444	0.553	0.531	0.208
600	0.487	0.443	0.555	0.529	0.145
650	0.495	0.449	0.540	0.532	0.136
700	0.500	0.452	0.544	0.529	0.107

Figure 4.3 illustrates one example each of the four potential outcomes: true positive (upper left), true negative (upper right), false positive (lower left), and false negative (lower right). Since the scales of defects are often small compared to the analysis window, the expanded portion of the images show the received update compared to the expected trajectory (blue line) and confidence bounds (red dotted lines).

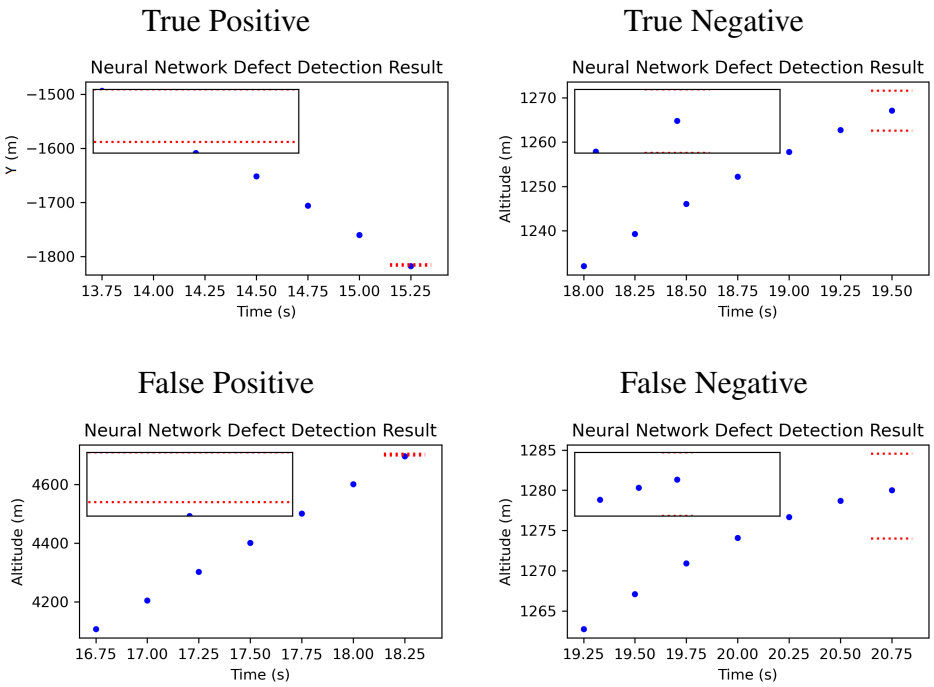

Figure 4.3: ANN defect detection result examples.

The success metrics defined in Section 3.3.1 are calculated and displayed in Table 4.8 for the ANN model with decay 0.0001 using a multiplier of 400 and a hidden size of 14 on the validation data.

Table 4.15: ANN success metrics.

Metric	**Value**	**LB**	**UB**
Recall	0.721	0.636	0.792
Precision	0.503	0.431	0.575
F_1	0.592	0.533	0.651

This method for detecting defects results in a recall of 72.1% (95% CI [63.6, 79.2]) and a precision of 50.3% (95% CI [43.1, 57.5]). This means that approximately 72% defects were detected and that roughly 50% of the detections were true positives, with the remainder being false

positives. The lower precision may or may not be acceptable depending on the use case. After a detection occurs, if the state update is thrown out and this has minimal effect on the DT, lower precision may be allowable. On the other hand, if an analyst manually reviews each detection, false positives may be very costly to adjudicate, and a decision will need to be made to balance precision and other measures.

Table 4.16: Defect severity breakdown for ANN.

Defect Severity	**True Positive**	**False Negative**	**Recall**	**LB**	**UB**
Low	12	19	0.387	0.237	0.562
Medium	61	18	0.772	0.668	0.851
High	25	1	0.962	0.811	0.993

Table 4.16 provides a performance breakdown by defect severity level for the ANN technique on the validation data data. It is not surprising to observe that the recall for low-severity defects is significantly less than for medium- and high-severity defects. It is also not surprising to see that High severity had the highest recall rate, although it was not significantly better than Medium when comparing the bounds.

4.3.1 Test Data

After demonstrating that ANN using a decay of 0.0001, a multiplier of 400, and a hidden size of 14 is a viable technique to detect defects in a sequential data feed, it is now applied to the set of test data. Reference Tables 4.17 and 4.18 for the test data defect detection results. This method results in a recall of 72.8% (95% CI [64.4, 79.9]) and a precision of 48.8% (95% CI [41.7, 55.8]). This means that approximately 73% defects were detected and that roughly 49% of all detections were true positives, with the remainder being false positives. The F_1 measure is 0.584 (95% CI [0.515, 0.653]). These results are comparable to the metrics observed in the validation data.

Table 4.17: Confusion matrix for ANN applied against the test data.

		Predicted		
		Positive	Negative	Total
Actual	Defect	99	37	136
	Good	104	19,607	19,711
	Total	203	19,644	19,847

Table 4.18: Summary of measures of success for ANN applied against the test data.

Success Measure	**Value**	**LB**	**UB**
Recall	0.728	0.644	0.799
Precision	0.488	0.417	0.558
F_1	0.584	0.515	0.653

Table 4.19 provides a performance breakdown by defect severity level for the ANN technique applied to the test data. Once again, low-severity defects were less likely to be detected than medium- and high-severity defects.

Table 4.19: Test data defect severity breakdown for ANN.

Defect Severity	**True Positive**	**False Negative**	**Recall**	**LB**	**UB**
Low	13	18	0.419	0.264	0.592
Medium	62	17	0.785	0.682	0.861
High	24	2	0.923	0.759	0.979

When comparing the results of the test data with the metrics of the validation data, the ANN defect detection technique is at least as effective in the independent test data as it was in the

validation data. This demonstrates the viability of this technique in detecting defects in sequential data for trajectories.

4.4 Meta-Algorithm

Following the research design in Section 3.3.5, a meta-algorithm model is applied to each trajectory update. Several decays and hidden sizes were tested to determine the best approach. Each combination of decays and hidden size was also tested with and without the time component. Of the top results for the meta-algorithm, the best F_1 values were 0.812 with the time component and 0.742 without the time component. Due to the higher F_1 value, the results of the meta-algorithm defect detection model with a time component are given in Table 4.20.

Table 4.20 displays a heat map with lower F_1 values in red and higher in green. It gives a quick look at how tuning the meta-algorithm model focused on the best parameters with a hidden size of 19 and decay of 0.01.

Table 4.20: Meta-algorithm results.

	Decay				
Hidden Size	**0.0001**	**0.001**	**0.01**	**0.1**	**1**
14	0.802	0.791	0.795	0.772	0.785
15	0.791	0.800	0.792	0.803	0.798
16	0.806	0.797	0.797	0.795	0.797
17	0.798	0.803	0.798	0.794	0.794
18	0.806	0.795	0.797	0.785	0.808
19	0.785	0.805	**0.812**	0.779	0.789
20	0.795	0.808	0.794	0.797	0.797
21	0.808	0.805	0.808	0.808	0.792
22	0.795	0.795	0.802	0.792	0.792
23	0.781	0.795	0.776	0.776	0.808

The success metrics defined in Section 3.3.1 are calculated and displayed in Table 4.21 for the meta-algorithm model using a hidden size of 19 and decay of 0.01 on the validation data.

Table 4.21: Meta-algorithm success metrics.

Metric	Value	LB	UB
Recall	0.785	0.705	0.849
Precision	0.841	0.763	0.898
F_1	0.812	0.759	0.865

This method for detecting defects results in a recall of 78.5% (95% CI [70.5, 84.9]) and a precision of 84.1% (95% CI [76.3, 89.8]). This means that approximately 79% defects were detected and that roughly 84% of the detections were true positives, with the remainder being false positives. The precision of this model is quite a bit better than that of the other three models individually.

Table 4.22: Defect severity breakdown for the meta-algorithm.

Defect Severity	True Positive	False Negative	Recall	LB	UB
Low	17	15	0.531	0.364	0.691
Medium	66	12	0.846	0.750	0.910
High	23	2	0.920	0.750	0.978

Table 4.22 provides a performance breakdown by defect severity level for the meta-algorithm technique on validation data. It is not surprising to observe that the recall for low-severity defects is significantly less than for medium- and high-severity defects.

4.4.1 Test Data

After demonstrating that a meta-algorithm using a hidden size of 19, decay of 0.01, and a time component is a viable technique to detect defects in a sequential data feed, it is now applied to

the set of test data. Reference Tables 4.23 and 4.24 for the test data defect detection results. This method results in a recall of 77.2% (95% CI [68.7, 83.9]) and a precision of 91.6% (95% CI [84.2, 95.8]). This means that approximately 77% defects were detected and that roughly 92% of all detections were true positives, with the remainder being false positives. The F_1 measure is 0.838 (95% CI [0.785, 0.892]). These results are comparable to the metrics observed in the validation data.

Table 4.23: Confusion matrix for the meta-algorithm applied against the test data.

		Predicted		
		Positive	Negative	Total
Actual	Defect	98	29	127
	Good	9	18,810	18,819
	Total	107	18,839	18,946

Table 4.24: Summary of measures of success for the meta-algorithm applied against the test data.

Success Measure	**Value**	**LB**	**UB**
Recall	0.772	0.687	0.839
Precision	0.916	0.842	0.958
F_1	0.838	0.785	0.892

Table 4.25 provides a performance breakdown by defect severity level for the meta-algorithm technique applied to the test data. Once again, low-severity defects were significantly less likely to be detected than medium and high-severity defects.

Table 4.25: Test data defect severity breakdown for the meta-algorithm.

Defect Severity	**True Positive**	**False Negative**	**Recall**	**LB**	**UB**
Low	19	15	0.559	0.395	0.711
Medium	60	9	0.870	0.770	0.930
High	19	5	0.792	0.595	0.908

When comparing the test data results against the metrics from the validation data, the meta-algorithm defect detection technique is at least as effective in the independent test data as it was in the validation data. This demonstrates the viability of this technique in detecting defects in sequential data for trajectories.

Chapter 5

Implications and Conclusions

5.1 Summary of Findings

This section provides a summary of the individual defect detection techniques presented in Chapter 4. The first success metric to examine is recall, which measures the proportion of actual defects correctly identified by each model. An accurate recall assessment is crucial in defect detection scenarios, as it reflects the model's ability to minimize false negatives and ensure that as many defects as possible are detected.

The results of the four models applied to the test data set are summarized in Table 5.1. This table presents the recall values obtained for each model, along with their corresponding lower and upper bounds. The inclusion of these bounds is important because it allows us to determine whether differences in recall between models are statistically significant or could have arisen due to random variation [80]. This statistical perspective provides a more rigorous basis for comparing model effectiveness and helps in identifying the most reliable defect detection approach.

Table 5.1: Summary of Recall.

Technique	**Recall**	**LB**	**UB**
Second-Order Polynomial Regression	0.773	0.690	0.839
ARIMA	**0.803**	0.723	0.865
Artificial Neural Network	0.728	0.644	0.799
Meta-Algorithm	0.772	0.687	0.839

The highest recall rate was 80.3% when using ARIMA. Regression and the meta-algorithm had similar recall rates at 77.3% and 77.2% respectively. The lowest recall came from the ANN at 72.8%. All of these recall values are within statistical significance, so more testing would need

to be completed to determine which actually provides the best recall. In a situation where false positives are not a concern, these data indicate that ARIMA might be the best model to use on this data type. However, these models were tuned for a balance between recall and precision, so a higher recall could be obtained, perhaps from a different model, if F_β was configured to give more weight to recall when training and evaluating the techniques.

These recall rates are quite high, especially when considering that only small differences were introduced as defects. Since larger differences are generally easier to detect, achieving such a high recall rate in this challenging scenario demonstrates the effectiveness of the detection methods. It is important to note that what constitutes a "good" recall rate can vary significantly depending on the specific application or use case. For some environments, even lower recall rates might be acceptable, while others demand near-perfect detection. In this case study, being able to identify between 70% and 80% defects when the deviations are subtle is a positive outcome. It indicates that the detection approaches are robust enough to identify most of the problematic instances even under challenging conditions, which is a promising result for practical deployment in real-world scenarios where small anomalies are critical to detect.

The next success metric is precision, which is the proportion of detections that actually have a defect. In other words, precision indicates how many of the identified defects are actually correct, reflecting the model's ability to minimize false positives. A high precision value signifies that the system is effective in avoiding incorrect identification of defects, which is essential for reducing unnecessary inspections or interventions.

To evaluate and compare the performance of the defect detection techniques, Table 5.2 summarizes the precision values obtained for each method. Alongside these raw precision scores, the table also presents the lower and upper confidence bounds, which provide insight into the variability and reliability of each measurement. These bounds allow us to determine whether the observed differences between techniques are statistically significant or may be attributed to sampling variability.

Table 5.2: Summary of Precision.

Technique	**Precision**	**LB**	**UB**
Second-Order Polynomial Regression	0.864	0.786	0.918
ARIMA	0.567	0.493	0.638
Artificial Neural Network	0.488	0.417	0.558
Meta-Algorithm	**0.916**	0.842	0.958

The highest precision rate was 91.6% when using the meta-algorithm, followed by regression at 86.4%, ARIMA at 56.7%, and ANN at 48.8%. The meta-algorithm and regression techniques were within statistical significance. Notably, these two techniques were significantly better than ARIMA and ANN. These models were tuned for a balance between recall and precision, so a higher precision could be obtained, perhaps from a different model, if the F_β was configured to give more weight to precision when training and evaluating the techniques.

Good precision may be even more subjective than good recall. Since there is a trade-off between precision and recall in classification problems, a practitioner will need to evaluate an acceptable false positive rate to obtain their desired recall. False positives can lead to more manual analysis and incorrect rejection of good data. Again, carefully choosing β for the F_β measure is a critical step in the model tuning process. The precision values for ARIMA and ANN are fine, but not impressive. However, the precision for regression and the meta-algorithm is quite good in an environment with small deviations from normal data.

Finally, the most important success metric is the F_1 measure, which is the harmonic mean between recall and precision. This metric provides a balanced evaluation of the performance of a model, offering a comprehensive assessment of its effectiveness in classification tasks. By balancing these two metrics, the F_1 score ensures that neither false positives nor false negatives disproportionately influence the evaluation.

Table 5.3 summarizes the F_1 scores for each technique examined in the study. These values provide a direct comparison of the overall effectiveness of the methods, with higher scores indi-

cating superior performance in both recall and precision. To account for variability and to provide a robust estimate of F_1 scores, the lower and upper bounds were derived using a bootstrapping approach. This method offers a nonparametric way to assess the reliability of the metric, ensuring that the reported bounds accurately represent the potential range of true F_1 scores under different data samples.

Table 5.3: Summary of F_1.

Technique	$\mathbf{F_1}$	**LB**	**UB**
Second-Order Polynomial Regression	0.816	0.765	0.870
ARIMA	0.665	0.605	0.725
Artificial Neural Network	0.584	0.515	0.653
Meta-Algorithm	**0.838**	0.785	0.892

The highest F_1 rate was 0.838 when using the meta-algorithm, followed by regression at 0.816, ARIMA at 0.665, and ANN at 0.584. The meta-algorithm and regression techniques were within statistical significance of each other. Notably, these two techniques were significantly better than ARIMA and ANN. As the balance between recall and precision and as the selected "decision" metric, F_1 scores will be the driving force behind model selection and decision making.

All of these techniques met the goal of a F_1 value of at least 0.5. As with recall and precision, a "good" F_1 is use case-dependent, but for an unbalanced data set with defects on such a small scale from good data, obtaining F_1 values above 0.8 is very promising. Both regression and the meta-algorithm performed well overall with good recall and great precision. Furthermore, the four techniques are proven to provide additional information above the three baseline models, whose F_1 were close to zero, indicating that no information was provided.

Using the F_1 measure, we can identify a best model. Although it was not statistically significant, the meta-algorithm using a time component proved slightly better than second-order polynomial regression alone. At least using the single F_1 values measured here on the test data set, this

appears to be the best model. However, another consideration is computational and model complexity. Since the regression model alone is about as effective in identifying defects and minimizing false positives in the data used in this case study, it might make more sense to use regression alone to detect defects. The technique is far more interpretable, simpler to understand, and less computationally intense than using two additional techniques and then passing those results through a meta-algorithm. In a real-world, real-time digital twin environment, taking computational resources away from the simulation could result in system lag. A practitioner weighing their options should consider whether the slightly better performance from using a meta-algorithm is worth the additional complexity.

5.1.1 RQ1: Detecting Defects in Sequential Inputs

Research Question 1: Can sequential input data with defects be detected in real time using second-order polynomial regression, ARIMA, and artificial neural networks?

With F_1 scores exceeding 0.5 for all three techniques, second-order polynomial regression, ARIMA, and ANN demonstrate the ability to detect defects in real-time applications involving sequential input data. The success metrics indicate that each model balances recall and precision sufficiently well, ensuring that most defects are identified without generating an excessive number of false positives. This performance underscores the models' effectiveness in environments where timely defect detection is critical.

All three models not only detected defects effectively, but also maintained acceptable levels of precision, indicating that their predictions are reliable and not overly prone to false alarms. The regression model in particular performed well to keep false positives low. The models' ability to add significant informational value suggests that they are capturing underlying patterns and features associated with defects, beyond what might be evident through manual inspection. Overall, this illustrates the potential of the models to improve data quality control through accurate real-time defect detection that can be incorporated into automated monitoring systems for DTs during execution.

5.1.2 RQ2: Using a Meta-Algorithm

Research Question 2: Do these defect detection techniques have better performance working together in a meta-algorithm?

The results demonstrate that using a meta-algorithm for defect detection yields performance that is at least on par with, and in some aspects superior to, the three individual techniques when used independently. The recall metric for the meta-algorithm was comparable to the three separate methods, which indicates that the meta-algorithm remains effective at capturing true positives. More notably, the meta-algorithm achieved a significantly higher precision compared to ARIMA and ANN, meaning it was better at minimizing false positives. The precision was also higher than the regression model, although it was not statistically significant. Furthermore, the meta-algorithm had the highest F_1 value, again significantly better than ARIMA and ANN, but not statistically significant from regression.

Overall, these findings suggest that integrating multiple techniques into a meta-algorithm can harness their individual strengths, leading to a more accurate and reliable defect detection system than relying on any single method alone. This approach offers a promising pathway to improve data quality. However, this case study relied heavily on data that were well suited to be predicted by second-order polynomial regression, so it could save computational resources and modeling complexity to use the regression model at the expense of some precision.

5.1.3 RQ3: Practical Use in Real-world Applications

Research Question 3: Are these techniques applicable to practitioners in the real world?

The success of the three techniques and the combined meta-algorithm demonstrate that these models are viable resources to practitioners who are concerned with the input data integrity for their DTs. Any user of digital twins should carefully consider their inputs and whether defect detection is necessary to keep their DT safe from anomalies. By validating the accuracy of the input data, researchers and practitioners can identify and exclude faulty or anomalous data before impacting the simulation outcomes. Sections 1.2 and 5.2 detail the pitfalls of defective inputs and the need

to detect them before they spoil the results, as well as providing a way to identify techniques that work in other situations, such as tuning β in the F_β measure to specific needs. These techniques and the concept of validating inputs in real time are directly applicable to real-world practitioners.

5.2 Discussion

While ideally all elements in a system of DTs have been verified and validated, it is often the case that not all elements are under the control of one of the DTs. These independent DTs must seek out methods to protect their output from defective inputs. This research has demonstrated the importance and impact of detecting these defects in real time. Using the techniques discussed here, defective trajectory updates from any source can be detected and handled.

Options for handling detected defects include ignoring the data and continuing the simulation with the prior data update or using the data as provided but flagging it for later analysis. Each has advantages and disadvantages that depend on the impact to the system if defects are allowed to pass through.

If a DT system takes the option to throw out detected bad data, it runs the risk of large gaps in acceptable updates. This is likely desirable for medium- and high-severity defects, as these defects are defined to have a greater influence on the system. This comes at the expense of having to propagate data forward internally, which could be itself a source of error as the propagated data deviates from truth. However, selectively discarding data does mean that defects cannot negatively influence simulation output. If, instead, detected defects are accepted, inputs containing bad data will be passed into the system at the expense of questionable outputs.

Independently of that decision, the system owner may choose to review or not review flagged potential defects. With a precision of 91.6% from the best method used in the case study, some time may still be spent analyzing false positives. The owner of a system would need to carefully weigh the cost and benefit of these reviews to decide whether it is worth the resources to manually analyze bad data.

As introduced in Section 3.3.1, one benefit of the F-measure when training a defect detection model is that the balance between recall and precision can be adjusted to the needs, appetite for risk, and available resources of the system owner. The generalized formula for the F_β-measure introduces β (see Equation 5.1), a parameter that represents how much more important the recall is than precision [67].

$$F_\beta = \frac{(1+\beta) \cdot precision \cdot recall}{(\beta^2 \cdot precision) + recall} \tag{5.1}$$

For a system owner who prioritizes catching as many defects as possible and has the resources to analyze false positives or a risk tolerance to allow them to be erroneously ignored by the digital twin, it may make sense to set β to be higher than one. On the other hand, the system owner may prefer higher precision because they do not have the resources to evaluate false positives, or they have a higher risk tolerance for defects slipping by. In this case, the model should be trained using a β less than one.

Another consideration of any sequential data defect detection technique is whether there is enough computational power to perform the detection while maintaining acceptable DT system performance. Some techniques are naturally more intense than others and, if the DT system needs to maintain real-time simulation, it may not be feasible to use more computationally heavy strategies. Fewer resources and tighter constraints may require lightweight techniques to meet the acceptability parameters. This is especially an issue when using a meta-learning strategy, which combines the results of two or more other techniques. Each of those defect detection models must be run independently before passing the results through more computation to reach a final result.

5.3 Suggestions for Future Research

Given the resultant high precision for the three independent techniques fed into a meta-algorithm as a defect detection technique, future research will focus on driving up recall to capture more anomalies while maintaining or improving precision ability. Increasing recall would minimize the number of defects that pass through the DT that can result in poor performance.

Areas of research could include robust defect detection techniques for a more general use case. Using second-order polynomial regression might work well for sequential data that is dependent on its most recent update, but some data may appear more random or may not follow a linear or quadratic trend. Robust techniques could be investigated to apply them generically to most or all sequential data regardless of their trend shape.

Both goals, increasing recall and use on general input data, could be accomplished with advanced machine learning tools. One such tool is the long short-term memory (LSTM) network, which is a sequential learning method that can predict the output associated with the next time steps and identify anomalous data (see Figure 5.1) [41]. This method holds promise, as it was designed for sequential data and holds some memory of the past, while learning which data should be remembered or forgotten for its predictions.

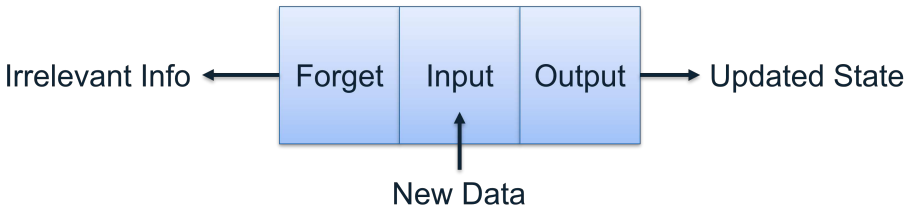

Figure 5.1: Basic LSTM workflow.

In certain scenarios, an important objective when developing models is to prioritize simplicity to enhance computational performance. Simplified models typically require fewer computational resources, such as processing power and memory, which allows faster training and defect detection times. This efficiency is especially crucial in real-time DTs or environments with limited hardware capabilities.

In addition, simple models are often more interpretable and transparent. When models are straightforward, such as regression or decision trees with few branches, it becomes easier to understand how inputs influence predictions. This interpretability may be vital in some domains where understanding the rationale behind a model's prediction is as important as the prediction itself. By reducing complexity, practitioners can also more effectively diagnose errors, identify biases,

and ensure the model is in line with expectations. Balancing simplicity with accuracy enables the development of models that are not only useful but also trustworthy and accessible to stakeholders who need to interpret or validate the results.

Section 2.2 provides several defect detection models for consideration. A practitioner would need to carefully consider which techniques are applicable and viable for their situation.

5.4 Conclusion

This work has proposed four methods for detecting defects in sequential data inputs used by digital twins during real-time simulation runs. Digital twins rely heavily on accurate and reliable data to mirror the behavior of physical systems, making input validation a critical component of simulation integrity. When data fed into a digital twin is flawed, it can lead to inaccurate predictions, misinterpretations, and potentially flawed decision-making. Validating digital twin inputs is essential to ensure the accuracy and reliability of simulation outcomes. By thoroughly validating the accuracy of input data, researchers and practitioners can identify and filter out defective or anomalous data before they influence the simulation results. This proactive approach helps maintain the fidelity of the digital twin, providing a reliable representation of the physical system it models.

The case study demonstrated the effectiveness of these validation methods. Using the proposed techniques, the defective data were successfully detected and isolated. By integrating these validation methods into a digital twin workflow, organizations can mitigate risks associated with data quality issues, improve the consistency of simulation outputs, and ultimately support better strategic and operational decisions.

In summary, this work underscores the importance of input validation in digital twin systems, demonstrating that effective detection of data defects not only improves simulation accuracy but also enhances the trustworthiness and utility of digital twins in real-world applications.

References

[1] Peter Hehenberger and David Bradley. *Mechatronic futures: Challenges and solutions for mechatronic systems and their designers*. Springer International Publishing, 1 2016.

[2] Richard Fujimoto, Conrad Bock, Wei Chen, Ernest Page, and Jitesh H Panchal Editors. *Research Challenges in Modeling and Simulation for Engineering Complex Systems*, pages 45–74. Springer, 2017.

[3] G. E. P. Box. *Robustness in the Strategy of Scientific Model Building*. Academic Press, 1979.

[4] Morten D. Skogen, Rubao Ji, Anna Akimova, Ute Daewel, Cecilie Hansen, Solfrid S. Hjøllo, Sonja M. Van Leeuwen, Marie Maar, Diego Macias, Erik Askov Mousing, Elin Almroth-Rosell, Sévrine F. Sailley, Michael A. Spence, Tineke A. Troost, and Karen Van de Wolfshaar. Disclosing the truth: Are models better than observations? *Marine Ecology Progress Series*, 680:7–13, 2021.

[5] Durk-Jouke van der Zee. An integrated conceptual modeling framework for simulation – linking simulation modeling to the systems engineering process. In *2012 Winter Simulation Conference*. IEEE, 2012.

[6] Guido A. Veldhuis, Nico M. de Reus, and Bas M.J. Keijser. Concept development for comprehensive operations support with modeling and simulation. *Journal of Defense Modeling and Simulation*, 17:99–116, 1 2020.

[7] Jason R Potts, Todd Griffith, Joseph Sharp, and Dan Allison. Subject matter expert-driven behavior modeling within simulation. *Industry Training, Simulation, and Education Conference*, 2019.

[8] Alexis Muller, Olivier Caron, Bernard Carré, and Gilles Vanwormhoudt. On some properties of parameterized model application. *Springer-Verlag Berlin Heidelberg*, 2005.

[9] IEEE. *IEEE Standard for Distributed Interactive Simulation— Application Protocols*. IEEE Computer Society, 2012.

[10] Jack P C Kleijnen. Verification and validation of simulation models. *European Journal of Operational Research*, 82:145–162, 1995.

[11] Alexander Kossiakoff, William N Sweet, Samuel J Seymour, and Steven M Biemer. *SYSTEMS ENGINEERING PRINCIPLES AND PRACTICE SECOND EDITION*. Wiley, 2011.

[12] Averill M. Law. How to build valid and credible simulation models. In *Proceedings - Winter Simulation Conference*, volume 2022-December, pages 1283–1295. Institute of Electrical and Electronics Engineers Inc., 2022.

[13] Adnan Khan, Martin Dahl, Petter Falkman, and Martin Fabian. Digital twin for legacy systems: Simulation model testing and validation. In *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, page 129. Institute of Electrical and Electronics Engineers, 2018.

[14] Robert G. Sargent. Verification and validation of simulation models. In *2010 Winter Simulation Conference*. IEEE, 2010.

[15] R Rebba, S Huang, Y Liu, and S Mahadevan. Statistical validation of simulation models. *Int. J. Materials and Product Technology*, 25:164–181, 2006.

[16] Mariam M.N. Aboelwafa, Karim G. Seddik, Mohamed H. Eldefrawy, Yasser Gadallah, and Mikael Gidlund. A machine-learning-based technique for false data injection attacks detection in industrial iot. *IEEE Internet of Things Journal*, 7:8462–8471, 9 2020.

[17] Hossein Darvishi, Domenico Ciuonzo, Eivind Rosón Eide, and Pierluigi Salvo Rossi. Sensor-fault detection, isolation and accommodation for digital twins via modular data-driven architecture. *IEEE Sensors Journal*, 21:4827–4838, 2 2021.

[18] Giovanni Lugaresi, Gianluca Aglio, Federico Folgheraiter, and Andrea Matta. Real-time validation of digital models for manufacturing systems: a novel signal-processing-based approach. In *15th International Conference on Automation Science and Engineering*, 2019.

[19] Salah Zidi, Tarek Moulahi, and Bechir Alaya. Fault detection in wireless sensor networks through svm classifier. *IEEE Sensors Journal*, 18:340–347, 1 2018.

[20] Cesare Alippi, Stavros Ntalampiras, and Manuel Roveri. Model-free fault detection and isolation in large-scale cyber-physical systems. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1:61–71, 2 2017.

[21] Jonghoon Lee, Jonghyun Kim, Ikkyun Kim, and Kijun Han. Cyber threat detection based on artificial neural networks using event profiles. *IEEE Access*, 7:165607–165626, 2019.

[22] Mete Ozay, Iñaki Esnaola, Fatos Tunay Yarman Vural, Sanjeev R. Kulkarni, and H. Vincent Poor. Machine learning methods for attack detection in the smart grid. *IEEE Transactions on Neural Networks and Learning Systems*, 27:1773–1786, 8 2016.

[23] Mohammad Esmalifalak, Lanchao Liu, Nam Nguyen, Rong Zheng, and Zhu Han. Detecting stealthy false data injection using machine learning in smart grid. *IEEE Systems Journal*, 11:1644–1652, 9 2017.

[24] Sheraz Naseer, Yasir Saleem, Shehzad Khalid, Muhammad Khawar Bashir, Jihun Han, Muhammad Munwar Iqbal, and Kijun Han. Enhanced network anomaly detection based on deep neural networks. *IEEE Access*, 6:48231–48246, 8 2018.

[25] Bahareh Pourbabaee, Nader Meskin, and Khashayar Khorasani. Sensor fault detection, isolation, and identification using multiple-model-based hybrid kalman filter for gas turbine engines. *IEEE Transactions on Control Systems Technology*, 24:1184–1200, 7 2016.

[26] Guillermo Heredia and Anibal Ollero. Detection of sensor faults in small helicopter uavs using observer/kalman filter identification. *Mathematical Problems in Engineering*, 2011, 2011.

[27] Alp Par. A scalable ai framework for smes and a novel metric: Proactive adaptation rate (par) journal of data analytics and artificial intelligence applications a scalable ai framework for smes and a novel metric: Proactive adaptation rate (par). *Article in Journal of Data Analytics and Artificial Intelligence Applications*, 2025.

[28] Jin Oh Kim, Young Min Kim, and Joo Yeoun Lee. Performance evaluation and improvement research of machine learning and deep learning models for industrial inverter failure prediction. *Journal of KOSSE*, 21:87–100, 2025.

[29] João Duarte, João Gama, and Albert Bifet. Adaptive model rules from data streams. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 10, 2016.

[30] Eduardo H.M. Pena, Marcos V.O. De Assis, and Mario Lemes Proença. Anomaly detection using forecasting methods arima and hwds. In *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, volume 0, pages 63–66. IEEE Computer Society, 7 2013.

[31] Asrul H. Yaacob, Ian K.T. Tan, Su Fong Chien, and Hon Khi Tan. Arima based network anomaly detection. In *2nd International Conference on Communication Software and Networks, ICCSN 2010*, pages 205–209, 2010.

[32] Vaia I. Kontopoulou, Athanasios D. Panagopoulos, Ioannis Kakkos, and George K. Matsopoulos. A review of arima vs. machine learning approaches for time series forecasting in data driven networks, 8 2023.

[33] Daniel J. Sargent. Comparison of artificial neural networks with other statistical approaches: Results from medical data sets. In *Conference on Prognostic Factors and Staging in Cancer Management: Contributions of Artificial Neural Networks and Other Statistical Methods*, volume 91, pages 1636–1642. John Wiley and Sons Inc., 4 2001.

[34] Muhammad Uzair and Noreen Jamil. Effects of hidden layers on the efficiency of neural networks. In *Proceedings - 2020 23rd IEEE International Multi-Topic Conference, INMIC 2020*. Institute of Electrical and Electronics Engineers Inc., 11 2020.

[35] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147, 11 2017.

[36] Tianyuan Lu, Lei Wang, and Xiaoyong Zhao. Review of anomaly detection algorithms for data streams, 5 2023.

[37] Waseem Ullah, Amin Ullah, Tanveer Hussain, Khan Muhammad, Ali Asghar Heidari, Javier Del Ser, Sung Wook Baik, and Victor Hugo C. De Albuquerque. Artificial intelligence of things-assisted two-stream neural network for anomaly detection in surveillance big video data. *Future Generation Computer Systems*, 129:286–297, 4 2022.

[38] Matthew. Ward and Giuseppe. Santucci. *IEEE Conference on Visual Analytics Science & Technology 2012 : Seattle, Washington, USA, 14-19 October 2012 : proceedings*. IEEE, 2012.

[39] Qingsong Wen, Jingkun Gao, Xiaomin Song, Liang Sun, Huan Xu, and Shenghuo Zhu. Robuststl: A robust seasonal-trend decomposition algorithm for long time series. In *The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, page 19. AAAI, 2019.

[40] Zhenwei Zhang, Ruiqi Wang, Ran Ding, and Yuantao Gu. Unravel anomalies: An end-to-end seasonal-trend decomposition approach for time series anomaly detection. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pages 5415–5419. Institute of Electrical and Electronics Engineers Inc., 2024.

[41] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 11 1997.

[42] Benjamin Lindemann, Benjamin Maschler, Nada Sahlab, and Michael Weyrich. A survey on anomaly detection for technical systems using lstm networks a survey on anomaly detection for technical systems using lstm networks. *Computers in Industry*, 131, 2021.

[43] Tolga Ergen and Suleyman Serdar Kozat. Unsupervised anomaly detection with lstm neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 31:3127–3141, 8 2020.

[44] H. D. Nguyen, K. P. Tran, S. Thomassey, and M. Hamad. Forecasting and anomaly detection approaches using lstm and lstm autoencoder techniques with the applications in supply chain management. *International Journal of Information Management*, 57, 4 2021.

[45] Zhaomin Chen, Chai Kiat Yeo, Bu Sung Lee, and Chiew Tong Lau. Autoencoder-based network anomaly detection. In *2018 Wireless Telecommunications Symposium (WTS)*. IEEE, 2018.

[46] Jinwon An and Sungzoon Cho. Variational autoencoder based anomaly detection using reconstruction probability. Technical report, SNU Data Mining Center, 2015.

[47] Chong Zhou and Randy C. Paffenroth. Anomaly detection with robust deep autoencoders. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume Part F129685, pages 665–674. Association for Computing Machinery, 8 2017.

[48] Mahmoud Said Elsayed, Nhien An Le-Khac, Soumyabrata Dev, and Anca Delia Jurcut. Network anomaly detection using lstm based autoencoder. In *Q2SWinet 2020 - Proceedings of the 16th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, pages 37–45. Association for Computing Machinery, Inc, 11 2020.

[49] Oleksandr I Provotar, Yaroslav M Linder, and Maksym M Veres. Unsupervised anomaly detection in time series using lstm-based autoencoders. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*. IEEE, 2019.

[50] Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18:77–95, 2002.

[51] Steven J. Simske. *Meta-algorithmics : patterns for robust, low cost, high quality systems*. Wiley-IEEE Press, 1st edition, 2013.

[52] Atsutoshi Kumagai, Tomoharu Iwata, Hiroshi Takahashi, and Yasuhiro Fujiwara. Meta-learning for robust anomaly detection. In *International Conference on Artificial Intelligence and Statistics (AISTATS) 2023*. Society for Artificial Intelligence and Statistics, 2023.

[53] Jhih Ciang Wu, Ding Jie Chen, Chiou Shann Fuh, and Tyng Luh Liu. Learning unsupervised metaformer for anomaly detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4349–4358. Institute of Electrical and Electronics Engineers Inc., 2021.

[54] Kaize Ding, Qinghai Zhou, Hanghang Tong, and Huan Liu. Few-shot network anomaly detection via cross-network meta-learning. *arXiv*, 2021.

[55] Hirak Mazumdar, Tae Hyeon Kim, Jong Min Lee, Euiseok Kum, Seungho Lee, Suho Jeong, and Bong Geun Chung. Sequential and comprehensive algorithm for fault detection in semiconductor sensors. *Applied Sciences (Switzerland)*, 11, 11 2021.

[56] Yap-Peng Tan and Tinku Acharya. A robust sequential approach for the detection of defective pixels in an image sensor. Technical report, Intel Corporation, 1998.

[57] Wenzhen Wang, Na Deng, and Binjie Xin. Sequential detection of image defects for patterned fabrics. *IEEE Access*, 8:174751–174762, 2020.

[58] Henry S Teng Kaihu Chen Stephen C-Y Lu and Donald-Lynch Blvd Knowledge-Based. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE, 1990.

[59] Hussain Nizam, Samra Zafar, Zefeng Lv, Fan Wang, and Xiaopeng Hu. Real-time deep anomaly detection framework for multivariate time-series data in industrial iot. *IEEE Sensors Journal*, 22:22836–22849, 12 2022.

[60] Andrea Castellani, Sebastian Schmitt, and Stefano Squartini. Real-world anomaly detection by using digital twin systems and weakly supervised learning. *IEEE Transactions on Industrial Informatics*, 17:4733–4742, 7 2021.

[61] Qinghua Xu, Shaukat Ali, and Tao Yue. Digital twin-based anomaly detection in cyber-physical systems. In *Proceedings - 2021 IEEE 14th International Conference on Software Testing, Verification and Validation, ICST 2021*, pages 205–216. Institute of Electrical and Electronics Engineers Inc., 4 2021.

[62] Dipanjan Sarkar, Raghav Bali, and Tushar Sharma. *Practical Machine Learning with Python*. Apress, 2018.

[63] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy, 9 2020.

[64] Giovani H Ceotto, Rodrigo N Schmitt, ; Guilherme, F Alves, ; Lucas, A Pezente, and Bruno S Carmo. Rocketpy: Six degree-of-freedom rocket trajectory simulator. *American Society of Civil Engineers*, 2021.

[65] N. Wiener and R. Paley. *Fourier Transforms in the Complex Domain*, volume 19. American Mathematical Society, 12 1934.

[66] Maria Isabel Ribeiro. Kalman and extended kalman filters: Concept, derivation and properties. *Institute for Systems and Robotics*, 2004.

[67] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27:861–874, 6 2006.

[68] Cyril Goutte and Eric Gaussier. *A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation*, pages 345–359. Springer, 2005.

[69] Lior M Burko and Richard H Price. Ballistic trajectory: parabola, ellipse, or what? *American Journal of Physics*, 2005.

[70] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning Data Mining, Inference, and Prediction*. Springer, 2008.

[71] K Doksum and J.-Y Koo. On spline estimators and prediction intervals in nonparametric regression. *Computational Statistics and Data Analysis*, 35:67–82, 2000.

[72] Akeyede Imam. On consistency of tests for stationarity in autoregressive and moving average models of different orders. *American Journal of Theoretical and Applied Statistics*, 5:146, 2016.

[73] Kun Il Park. *Fundamentals of probability and stochastic processes with applications to communications*. Springer International Publishing, 11 2017.

[74] Tomasz Szandala. *Studies in Computational Intelligence: Bio-inspired Neurocomputing*, volume 903, page 203. Springer, 2021.

[75] Kensuke Nakamura and Byung Woo Hong. Adaptive weight decay for deep neural networks. *IEEE Access*, 7:118857–118865, 2019.

[76] Xue Ying. An overview of overfitting and its solutions. In *Journal of Physics: Conference Series*, volume 1168. Institute of Physics Publishing, 3 2019.

[77] Stephen Allwright. What is a good f1 score and how do i interpret it?, 7 2025.

[78] Cuthbert Ruseruka, Judith Mwakalonge, Gurcan Comert, Saidi Siuhi, Frank Ngeni, and Kristin Major. Pavement distress identification based on computer vision and controller area network (can) sensor models. *Sustainability (Switzerland)*, 15, 4 2023.

[79] Marketa Ciharova, Khadicha Amarti, Ward van Breda, Martin J. Gevonden, Sina Ghassemi, Annet Kleiboer, Christiaan H. Vinkers, Milou S. C. Sep, Sophia Trofimova, Alexander C. Cooper, Xianhua Peng, Mieke Schulte, Eirini Karyotaki, Pim Cuijpers, and Heleen Riper. Machine-learning detection of stress severity expressed on a continuous scale using acoustic, verbal, visual, and physiological data: lessons learned. *Frontiers in Psychiatry*, 16, 6 2025.

[80] NIST/SEMATECH. What is the relationship between a test and a confidence interval?, 4 2012.

Appendix A

Acronyms

Acronym	Definition
ANN	Artificial neural network
AR	Autoregression
ARIMA	Autoregressive integrated moving average
CI	Confidence interval
DIS	Distributed Interactive Simulation
DoD	Department of Defense
DT	Digital twin
FN	False negative
FP	False positive
HMM	Hidden Markov models
IEEE	The Institute of Electrical and Electronics Engineers
LB	Lower bound
LOESS	Locally estimated scatterplot smoothing
LSTM	Long short-term memory
M&S	Modeling and simulation
MA	Moving Average
NaN	Not a number
NPV	Negative predictive value
PDU	Protocol data units
RCS	Radar cross section
RNG	Random number generator
SE	Standard error

STL	Seasonal and trend decomposition using LOESS
SVM	Support vector machine
TN	True negative
TP	True positive
UB	Upper bound
V&V	Verification and validation

Appendix B

Source Code and Data Location

All Python and R code with the data used for this analysis is available on Gitlab: https://gitlab.com/n.brown1735/a-framework-for-detecting-defects-in-sequential-inputs-to-modeling-and-simulation-using-machine-learning-techniques

Appendix C

Selected Source Code Snippets

```
"""
Main.py

Notes:
    The purpose of the Main module is to generate all data, clean it, modify the
    experimental data, execute tests in R to detect defects, and graph results.

Use:
    Run each line step by step

IMPORTANT:
    I have designed this file to be run one or two lines at a time. Running
    everything at once is ill-advised. Certainly you will not need to
    generate the data each time you run the file and just read the saved data.

"""

##### Debug Steps #####
# import traceback
# import logging
#try:
#    <whatever line you want to debug>
#except Exception:
#    logging.error(traceback.format_exc())


##### Set Path #####
# Must be set by user to top level of project
source="<YOUR_PATH>"


##### Add source code to path #####
import sys
source=source.rstrip('//') # Remove final / if it exists
sys.path.append(source + "/code/dataGeneration/")
sys.path.append(source + "/code/dataSummaryAndPlots/")
sys.path.append(source + "/code/examplesAndTests/")
sys.path.append(source + "/code/dataAnalysis/")
sys.path.append(source + "/code/supportTools/")


##### Generate Data #####
# NOTE: Skip to Read Data section, if already generated

# Generate baseline data
from generateData import generateData
data = generateData(quant=50)

# Introduce defects
from generateExperimentData import modifyData
data = modifyData(data, expProp=0.667, modProp=0.01)

# Measure severity of defects using Kalman filter
from defectSeverity import defectSeverity
data = defectSeverity(data, sigma_m=1)


##### Save Data #####
# Saving as a CSV file for easy manual validation
```

```
data.to_csv(source + "/data/data.csv", index=False)


##### Read Data #####
from pandas import read_csv
data = read_csv(source + "/data/data.csv", index_col=False)


##### Exploratory data summaries and plots #####

# Show Kalman filter predictions and error bounds
from kalmanFilterPlots import *

# Summary of base data
from baseDataStatistics import baseDataStatistics
baseDataStatistics(data, source + "/data/")

# Base data plots
from baseDataPlots import *
defectExamplePlots(source + "/plots/")
flightPlots(data, 'Flight00001', source + "/plots/")


##### Data Analysis #####
# At this point, we let R do the heavy statistical lifting. If I can ever get
# rpy2 to work on my computer, perhaps these steps can be called directly from
# here in the future, but the future is not now. If you like to live life on the
# edge (and have R installed), you can just blindly run the line below.
# Otherwise, navigate to Main.R and step through that process.
import subprocess
subprocess.call("Rscript " + source + "/code/Main.R", shell=True)
# Really, really don't recommend this at all


##### Analysis summaries and plots #####
# Polynomial plots
from polynomialPlots import *
polynomialExamplePlots(source + "/plots/")
polynomialPlots(data, 'Flight00002', 100, source + "/plots/")   #True Positive
polynomialPlots(data, 'Flight00001', 3, source + "/plots/")     #True Negative
polynomialPlots(data, 'Flight00004', 21.5, source + "/plots/")  #False Positive
polynomialPlots(data, 'Flight00002', 43, source + "/plots/")    #False Negative

# ARIMA plots
from arimaPlots import *
arimaPlots(data, 'Flight00002', 171.75, source + "/plots/")  #True Positive
arimaPlots(data, 'Flight00001', 4.5, source + "/plots/")     #True Negative
arimaPlots(data, 'Flight00003', 34.5, source + "/plots/")    #False Positive
arimaPlots(data, 'Flight00009', 32, source + "/plots/")      #False Negative

# Neural Net plots
from nnetPlots import *
from pandas import read_csv
dataVal = read_csv(source + "/dataValidation/dataValidation.csv", index_col=False)
#nnetPlots(dataVal, 'Flight00002', 15.25, source + "/plots/")  #True Positive (plot Y, not Z)
nnetPlots(dataVal, 'Flight00001', 19.5, source + "/plots/")     #True Negative
nnetPlots(dataVal, 'Flight00011', 18.25, source + "/plots/")    #False Positive
nnetPlots(dataVal, 'Flight00001', 20.75, source + "/plots/")      #False Negative



##### Create test data #####
from generateTestData import generateTestData
dataTest = generateTestData()
dataTest.to_csv(source + "/dataTest/dataTest.csv", index=False)

# Summary of base data
from baseDataStatistics import baseDataStatistics
```

```
baseDataStatistics(dataTest, source + "/dataTest/")


##### END OF FILE #####
```

```
# """
# Main.R
#
# Notes:
#     The purpose of the Main module is to use the data generated by Main.py and
#
#
# Use:
#     Run each line step by step
#
# IMPORTANT:
#     Unlike Main.py, this file could be run all at once if desired, but that is
#     no fun. Run each line one or two at a time to manage run time.
#
# """


##### Debug Steps #####
# Activate option below to automatically print a backtrace with an error
options(error = traceback)


##### Set Path #####
# Change path to top level code location
path <- "<YOUR_PATH>"
setwd(path)


##### Parallel Processing #####
# Set n.cores to be the number of cores dedicated to parallel processing
# Recommend leaving a core or two available for other tasks
# Set to 1 if encountering unexpected errors.
n.cores <- 12
source(paste(getwd(), "/code/supportTools/parallelProcessing.R", sep=""))


##### Read Data #####
df <- read.csv(paste(getwd(), "/data/data.csv", sep=""))


##### Defect Detection #####

# Baseline Models
source(paste(getwd(), "/code/dataAnalysis/baselineModels.R", sep=""))
df <- baselineDefectDetection(df)

# Polynomial Regression
source(paste(getwd(), "/code/dataAnalysis/polynomialRegression.R", sep=""))
    # Just run the best model
    df <- polynomialRegression(df, 0.99, 20.0, 6)
    # Or loop over some values, which will take a while...
    # Seriously, it is going to take a long time, a day or two, I don't know...
    for (i in list(1, 2, 5, 10, 20, 50, 100, 150)) { #iterate through multiplier
        for (j in list(4, 5, 6, 7, 8, 9, 10, 11, 12)) { #iterate through window size
            df <- polynomialRegression(df, 0.99, i, j)
        }
    }

# ARIMA
source(paste(getwd(), "/code/dataAnalysis/arima.R", sep=""))
    # Just run the best model
    df <- arimaFunction(df, 0.99, 600, 2, 2, 0, 8)
    # Or loop over some values, which will take a while...
    # Seriously, it is going to take a long time, a day or two, I don't know...
    for (i in list(50, 100, 150, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1200, 1400, 1600))
    ↪  { #iterate through multiplier
        for (j in list(7, 8, 9)){ #iterate through window size
            df <- arimaFunction(df, 0.99, i, 2, 2, 0, j)
```

```
        }
    }

# Neural Network
# Does not return anything, all data is saved to files internally
source(paste(getwd(), "/code/dataAnalysis/neuralNet.R", sep=""))
    dfTrain <- read.csv(paste(getwd(), "/data/data.csv", sep=""))
    dfVal <- read.csv(paste(getwd(), "/dataValidation/dataValidation.csv", sep=""))
    dfTest <- read.csv(paste(getwd(), "/dataTest/dataTest.csv", sep=""))
    # Just run the best model
    neuralNet(dfTrain, dfVal, dfTest, 400.0, 14, 0.0001, 1)
    # Or loop over some values, which will take a while...
    mult <- c(1, 2, 5, 10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600,
    ↪  650, 700, 750, 800, 850, 900, 950, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800,
    ↪  1900, 2000, 2200, 2400, 2600, 2800, 3000) #iterate through some multipliers
    for (i in list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
    ↪  23, 24, 25)) { # iterate through some hidden layer sizes
        for (j in list(0, 0.0001, 0.001, 0.01, 0.1, 1)) { #iterate through some delay
            neuralNet(dfTrain, dfVal, dfTest, mult, i, j, 0)
        }
    }

# Meta Algorithm
# Does not return anything, all data is saved to files internally
source(paste(getwd(), "/code/dataAnalysis/metaAlgorithm.R", sep=""))
    dfTrain <- read.csv(paste(getwd(), "/data/data.csv", sep=""))
    dfVal <- read.csv(paste(getwd(), "/dataValidation/dataValidation.csv", sep=""))
    dfTest <- read.csv(paste(getwd(), "/dataTest/dataTest.csv", sep=""))
    # Just run the best model
    metaAlgorithm(dfTrain, dfVal, dfTest, 19, 0.1, 1)
    # Or loop over some values, which will take a while, but not as long as previous two...
    for (i in list(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
    ↪  22, 23, 24, 25)) { # iterate through some hidden layer sizes
        for (j in list(0, 0.0001, 0.001, 0.01, 0.1, 1)) { #iterate through some delay
            metaAlgorithm(dfTrain, dfVal, dfTest, i, j, 0)
        }
    }


##### Save Data #####
write.csv(df, paste(getwd(), "/data/data.csv", sep=""), row.names=FALSE)


##### Run models against test data

    # !!! NOTE !!!: Before running anything here, go to logResultsSummary.R and
    # change the location <- "/dataTest" variable. Don't forget to change it back!
    # I'll parameterize it someday...

    # Polynomial Regression
    dfTest <- read.csv(paste(getwd(), "/dataTest/dataTest.csv", sep=""))
    source(paste(getwd(), "/code/dataAnalysis/polynomialRegression.R", sep=""))
    dfTest <- polynomialRegression(dfTest, 0.99, 20.0, 6)
    write.csv(dfTest, paste(getwd(), "/dataTest/dataTest.csv", sep=""), row.names=FALSE)

    # ARIMA
    dfTest <- read.csv(paste(getwd(), "/dataTest/dataTest.csv", sep=""))
    source(paste(getwd(), "/code/dataAnalysis/arima.R", sep=""))
    dfTest <- arimaFunction(dfTest, 0.99, 600, 2, 2, 0, 8)
    write.csv(dfTest, paste(getwd(), "/dataTest/dataTest.csv", sep=""), row.names=FALSE)

    # Neural Network
    # No need to do anything for Neural Net, it handles training, validation, and test
    ↪  internally.


##### END OF FILE #####
```

```
"""
generateData.py

Notes:
    The purpose of this function is to generate all data for the training and
    testing of defect detection models.

Use:
    generateData(quant)

        Arguments:
        quant   -- the number of trajectories desired (default 1)

    Returns data structure with control trajectories

"""

from generateTrajectory import generateTrajectory
from repeatableRandom import *
from tqdm import tqdm
from math import floor
from numpy import arange
from pandas import DataFrame


# Generate trajectories and prune data to reduce total size
def generateData(quant=1, seed=0):
    # Validate inputs
    if not (1 <= quant):
        ValueError('The number of flights should be greater than or equal to 1')

    # Reset random number generator
    repRandReset(seed)

    # Initialize data structure
    data = {
        "flightID": [],          #Unique ID for each flight
        "flightDefect": [],      #Flag to mark a flight with at least one defect
        "rowDefect": [],         #Flag to mark a row with at least one defect
        "defectSeverity": [],    #Severity of the defect, if there is one
        "time": [],              #Flight time (s)
        "x": [],                 #x-axis position (m), may contain defect
        "xClean": [],            #x-axis (m), value before defect introduced
        "xKalman": [],           #x-axis (m), predicted from Kalman filter
        "xKalVar": [],           #x-axis variance of prediction from Kalman filter
        "y": [],                 #y-axis position (m), may contain defect
        "yClean": [],            #y-axis (m), value before defect introduced
        "yKalman": [],           #y-axis (m), predicted from Kalman filter
        "yKalVar": [],           #y-axis variance of prediction from Kalman filter
        "z": [],                 #z-axis position (m), may contain defect
        "zClean": [],            #z-axis (m), value before defect introduced
        "zKalman": [],           #z-axis (m), predicted from Kalman filter
        "zKalVar": [],           #z-axis variance of prediction from Kalman filter
        "vx": [],                #x-axis velocity (m/s), may contain defect
        "vxClean": [],           #x-axis velocity (m/s), value before defect introduced
        "vy": [],                #y-axis velocity (m/s), may contain defect
        "vyClean": [],           #y-axis velocity (m/s), value before defect introduced
        "vz": [],                #z-axis velocity (m/s), may contain defect
        "vzClean": [],           #z-axis velocity (m/s), value before defect introduced
        "ax": [],                #x-axis acceleration (m/s^2)
        "ay": [],                #y-axis acceleration (m/s^2)
        "az": []                 #z-axis acceleration (m/s^2)
    }

    # Generate the trajectories and clean the data to be more manageable
    for i in tqdm(range(quant)):
        # Random inputs for flight
        lat = repRandDec(-90, 90)
```

```
        long = repRandDec(-180, 180)
        inc = repRandDec(45, 85)
        heading = repRandDec(0, 360)
        burnout = repRandDec(4, 45)

        # Generate flight
        flight = generateTrajectory(lat, long, inc, heading, burnout)

        # Convert flight object into data structure
        flightName = 'Flight' + str(i+1).rjust(5, '0')
        timeIncrements = list(arange(0, floor(flight.t*4)/4+0.001, 0.25))
        timeIncrements.append(round(flight.t, 2)) #include final time and state of flight

        flight.postProcess()
        data["flightID"].extend([flightName] * len(timeIncrements))
        # Flight and data defects and detections will be set to something != 0 later
        data["flightDefect"].extend([0] * len(timeIncrements))
        data["rowDefect"].extend([0] * len(timeIncrements))
        data["defectSeverity"].extend(['None'] * len(timeIncrements))
        data["time"].extend(timeIncrements)
        data["x"].extend(flight.x(timeIncrements))
        data["xClean"].extend(flight.x(timeIncrements))
        data["xKalman"].extend([0] * len(timeIncrements))
        data["xKalVar"].extend([0] * len(timeIncrements))
        data["y"].extend(flight.y(timeIncrements))
        data["yClean"].extend(flight.y(timeIncrements))
        data["yKalman"].extend([0] * len(timeIncrements))
        data["yKalVar"].extend([0] * len(timeIncrements))
        data["z"].extend(flight.z(timeIncrements))
        data["zClean"].extend(flight.z(timeIncrements))
        data["zKalman"].extend([0] * len(timeIncrements))
        data["zKalVar"].extend([0] * len(timeIncrements))
        data["vx"].extend(flight.vx(timeIncrements))
        data["vxClean"].extend(flight.vx(timeIncrements))
        data["vy"].extend(flight.vy(timeIncrements))
        data["vyClean"].extend(flight.vy(timeIncrements))
        data["vz"].extend(flight.vz(timeIncrements))
        data["vzClean"].extend(flight.vz(timeIncrements))
        data["ax"].extend(flight.ax(timeIncrements))
        data["ay"].extend(flight.ay(timeIncrements))
        data["az"].extend(flight.az(timeIncrements))


    # Convert to pandas DataFrame
    data = DataFrame(data)
    return data
```

```
"""
generateTrajectory.py

Notes:
    The purpose of this function is to generate a trajectory based on a set of
    inputs defined below. Rocketpy is the generator of the data.

Use:
    generateTrajectory(lat, long, inc, heading, burnout)

        Arguments:
        lat     -- Latitude degrees in decimal format [-90, 90] (default 0)
        long    -- Longitude degrees in decimal format [-180, 180] (default 0)
        inc     -- Inclination degrees relative to ground [45, 85] (default 85)
        heading -- Heading degrees relative to North [0, 360] (default 0)
        burnout -- Time to burnout of fuel in seconds [4, 45] (default 4)

    Returns a dictionary object

"""

from rocketpy import Environment, Rocket, SolidMotor, Flight
import sys

def generateTrajectory(lat=0, long=0, inc=85, heading=0, burnout=4):

    # Validate arguments
    if not (-90 <= lat <= 90):
        ValueError('Latitude must be between -90 and 90 degrees')

    if not (-180 <= long <= 180):
        ValueError('Longitude must be between -180 and 180 degrees')

    if not (45 <= inc <= 85):
        ValueError('Inclination must be between 45 and 90 degrees relative to ground')

    if not (0 <= heading <= 360):
        ValueError('Heading must be between 0 and 360 degrees relative to North')

    if not (4 <= burnout <= 45):
        ValueError('Burnout must be between 4 and 45 seconds')


    # Begin setup of Flight

    Env = Environment(
        railLength=5.2,
        latitude=lat,
        longitude=long,
        elevation=0,
        date=(2023, 1, 22, 12) # May as well fix the date
    )

    Env.setAtmosphericModel(type='StandardAtmosphere', file='GFS')

    Pro75M1670 = SolidMotor(
        thrustSource=1000, #"./my_env/Lib/data/motors/Cesaroni_M1670.eng",
        burnOut=burnout,
        grainNumber=5,
        grainSeparation=5/1000,
        grainDensity=1815,
        grainOuterRadius=33/1000,
        grainInitialInnerRadius=15/1000,
        grainInitialHeight=120/1000,
        nozzleRadius=33/1000,
        throatRadius=11/1000,
        interpolationMethod='linear'
    )
```

```
Calisto = Rocket(
    motor=Pro75M1670,
    radius=127/2000,
    mass=19.197-2.956,
    inertiaI=6.60,
    inertiaZ=0.0351,
    distanceRocketNozzle=-1.255,
    distanceRocketPropellant=-0.85704,

    ↪  powerOffDrag=sys.path[len(sys.path)-1]+"../../rocketPyData/calisto/powerOffDragCurve.csv",
    powerOnDrag=sys.path[len(sys.path)-1]+"../../rocketPyData/calisto/powerOnDragCurve.csv"
)

Calisto.setRailButtons([0.2, -0.5])

NoseCone = Calisto.addNose(length=0.55829, kind="vonKarman", distanceToCM=0.71971)

FinSet = Calisto.addFins(4, span=0.100, rootChord=0.120, tipChord=0.040,
↪  distanceToCM=-1.04956)

Tail = Calisto.addTail(topRadius=0.0635, bottomRadius=0.0435, length=0.060,
↪  distanceToCM=-1.194656)

# Fly rocket using parameters above
outputFlight = Flight(rocket=Calisto, environment=Env, inclination=inc, heading=heading)

return outputFlight
```

```
"""
generateData.py

Notes:
    The purpose of this function is to modify the control data for the training
    and testing of defect detection models. This includes randomly inserting
    defects, but does not include

Use:
    modifyData(data, expProp, modProp)

        Arguments:
        data    -- Pandas DataFrame object containing flight data
        expProp -- Proportion between 0 and 1 of experimental trajectories versus
                   control, results round(quant*expProp) experimentals (default 0)
        modProp -- Proportion between 0 and 1 of data in an experimental
                   trajectory that is modified, should be small (default 0)

    Returns data structure with control and experimental trajectories

"""

from repeatableRandom import *
from tqdm import tqdm
from math import sqrt, log, cos, pi
from collections import Counter
from pandas import unique

# Start by defining a few strategies for modifying the data
# Gaussian (normal) distribution
def gauss(stdev=1):
    return stdev * sqrt(-2 * log(repRand()) ) * cos(2 * pi * repRand())

# Uniform distribution
def unif(spread=1):
    return repRandDec(-1 * spread, spread)

# Randomly modify some of the control data to be experimental data
def modifyData(data, expProp=0, modProp=0):
    # Validate arguments
    if not (0 <= expProp <= 1):
        ValueError('Proportion of experimental flights should be between 0 and 1')

    if not (0 <= modProp <= 1):
        ValueError('Proportion of modified flight data should be between 0 and 1')

    # Reset repeatable RNG
    repRandReset()

    # Determine number of experimental and control trajectories
    quant = len( unique(data.flightID) )
    exp = round( quant * expProp )
    ctrl = quant - exp
    print('Out of ' + str(quant) + ' total entries, there will be ' +
          str(ctrl) + ' control data and ' + str(exp) + ' experimental data. \n')

    # Determine which flights will be experimental
    expFlights = [( 'Flight' + str(x).rjust(5, '0') )
                  for x in (choice(quant, exp, False) + 1)]

    # Loop through experimental data and modify specific data points
    for flight in tqdm(expFlights):
        # Set flightDefect to 1 for this experimental flightID
        data.loc[data.flightID == flight, 'flightDefect'] = 1

        # Determine which rows will be experimental
        numRows = len( data.loc[data.flightID == flight] )
        numExpRows = max( 1, round(numRows*modProp) ) # Always select at least 1
```

```
    expRows = choice(numRows, numExpRows, False)
    minIndex = min(data.loc[data.flightID == flight].index) # Need for adding to expRows

    # Loop through expRows and modify data per described techniques
    for row in expRows:
        # First, set the defect row flag
        data.iloc[minIndex + row, data.columns.get_loc("rowDefect")] = 1

        # Then set some defects
        data.iloc[minIndex + row, data.columns.get_loc("x")]  += gauss(0.75)
        data.iloc[minIndex + row, data.columns.get_loc("y")]  += gauss(0.75)
        data.iloc[minIndex + row, data.columns.get_loc("z")]  += gauss(0.75)
        #data.iloc[minIndex + row, data.columns.get_loc("vx")] += gauss(100)
        #data.iloc[minIndex + row, data.columns.get_loc("vy")] += gauss(100)
        #data.iloc[minIndex + row, data.columns.get_loc("vz")] += gauss(100)
        #TODO probably will need to tune the gauss() value to give even number of
        ↪  low/med/high


return data
```

```
"""
defectSeverity.py

Notes:
    The purpose of this module is to assign a severity to the defective data. We
    will use a Kalman Filter, which is descibed in the paper.

    Defect Severity
    Low  -> Most elements in a simulation will not behave poorly with defect
    Med  -> Some elements will behave poorly with medium defects
    High -> Most or all elements will behave poorly with high defects

Use:
    defectSeverity(data, sigma_m)

        Arguments:
        data    -- Pandas DataFrame object containing flight data
        sigma_m -- Measurement st dev for Kalman filter, but used as a measure of model
↪   sensitivity

    Returns data structure with control and experimental trajectories adding
    the defect severity for all defects

"""

from kalmanFilter import *
from tqdm import tqdm
from numpy import array, sqrt

def defectSeverity(data, sigma_m):
    # Get list of flights in data
    flights = data.flightID.unique().tolist()

    # Loop through flights and pass through Kalman filter
    for flight in tqdm(flights):
        # Determine flight information to loop through
        numRows = len( data.loc[data.flightID == flight] )
        minIndex = min(data.loc[data.flightID == flight].index) # Need for adding to row

        # Initialize Kalman filter
        x, P = initKalman()
        data.iloc[minIndex, data.columns.get_loc("xKalman")] = x[0,0]
        data.iloc[minIndex, data.columns.get_loc("yKalman")] = x[3,0]
        data.iloc[minIndex, data.columns.get_loc("zKalman")] = x[6,0]
        data.iloc[minIndex, data.columns.get_loc("xKalVar")] = P[0,0]
        data.iloc[minIndex, data.columns.get_loc("yKalVar")] = P[1,1]
        data.iloc[minIndex, data.columns.get_loc("zKalVar")] = P[2,2]
        sigma_a=0.2

        # Loop through rows and include Kalman filter predictions
        for row in range(1, numRows):
            # Get measurement
            z = array([[data.iloc[minIndex + row, data.columns.get_loc("x")]],
                       [data.iloc[minIndex + row, data.columns.get_loc("y")]],
                       [data.iloc[minIndex + row, data.columns.get_loc("z")]]])

            # Get timestep
            deltaT = data.iloc[minIndex + row, data.columns.get_loc("time")] - \
                     data.iloc[minIndex + row - 1, data.columns.get_loc("time")]

            # Step Kalman filter and store in data
            x, P = stepKalman(x, P, deltaT, sigma_a, sigma_m, z)
            data.iloc[minIndex + row, data.columns.get_loc("xKalman")] = x[0,0]
            data.iloc[minIndex + row, data.columns.get_loc("yKalman")] = x[3,0]
            data.iloc[minIndex + row, data.columns.get_loc("zKalman")] = x[6,0]
            data.iloc[minIndex + row, data.columns.get_loc("xKalVar")] = P[0,0]
            data.iloc[minIndex + row, data.columns.get_loc("yKalVar")] = P[1,1]
            data.iloc[minIndex + row, data.columns.get_loc("zKalVar")] = P[2,2]
```

```
# Assign severity to each row that is flagged as modified
columnList = ['x','y','z'] # List of columns we care about with defects

data['defectSeverity'] = 'None' # Initialize all defectSeverity values
data.loc[data.rowDefect==1, 'defectSeverity'] = 'Low' # All defects are at least low


for column in data.loc[:,columnList].columns:
    kalVarCol = column + 'KalVar'
    cleanCol = column + 'Clean'

    # Medium defects are at least one standard deviation from expectation (Kalman)
    data.loc[(data.rowDefect==1) & (data.defectSeverity != "High") &
    ↪  (abs(data[column]-data[cleanCol])>sqrt(data[kalVarCol])), 'defectSeverity'] = 'Med'

    # High defects are at least two standard deviations from expectation (Kalman)
    data.loc[(data.rowDefect==1) &
    ↪  (abs(data[column]-data[cleanCol])>2*sqrt(data[kalVarCol])), 'defectSeverity'] =
    ↪  'High'


return data
```

```
"""
kalmanFilter.py

Notes:
    The purpose of this module is to use a Kalman filter for motion to smooth
    the state updates and predict the next state and covariance. This will be
    used to assign low, medium, and high severity to defects.

Use:
    See each function below

"""

from numpy import array, eye, linalg

# Function to initialize Kalman filter
def initKalman():
    '''
        The purpose of this function is to initialize the Kalman filter. The
        initial state (x) is assumed to be zeros because all flight data uses
        the origin as the starting point and the initial covariance (P) is
        assumed to be quite small at first due to our knowledge of the flight
        starting point and initial velocity.

        Outputs:
            x       -- Initial assumed state [x, vx, ax, y, vy, ay, z, vz, az]
            P       -- Initial assumed estimate uncertainty
    '''
    x = array([[   0,    0,    0,    0,    0,    0,    0,    0,    0]]).T
    P = array([[100000,    0,    0,    0,    0,    0,    0,    0,    0],
               [   0, 100000,    0,    0,    0,    0,    0,    0,    0],
               [   0,    0, 100000,    0,    0,    0,    0,    0,    0],
               [   0,    0,    0, 1000,    0,    0,    0,    0,    0],
               [   0,    0,    0,    0, 1000,    0,    0,    0,    0],
               [   0,    0,    0,    0,    0, 1000,    0,    0,    0],
               [   0,    0,    0,    0,    0,    0, 100,    0,    0],
               [   0,    0,    0,    0,    0,    0,    0, 100,    0],
               [   0,    0,    0,    0,    0,    0,    0,    0, 100]])

    return x, P


# Function to step through measurements and update the Kalman filter
def stepKalman(x, P, deltaT, sigma_a, sigma_m, z):
    '''
        Inputs:
            x       -- Current state
            P       -- Current estimate uncertainty
            deltaT  -- Time difference since current state
            sigma_a -- Acceleration st dev
            sigma_m -- Measurement st dev
            z       -- Measurement

        Outputs:
            x       -- Next state
            P       -- Next estimate uncertainty

    '''

    # State transition matrix
    F = array([[1, deltaT, 0.5*(deltaT**2), 0,      0,               0, 0,      0,
    ↪  0],
               [0,      1,          deltaT, 0,      0,               0, 0,      0,
               ↪  0],
               [0,      0,               1, 0,      0,               0, 0,      0,
               ↪  0],
               [0,      0,               0, 1, deltaT, 0.5*(deltaT**2), 0,      0,
               ↪  0],
```

```
            [0,      0,                0, 0,      1,          deltaT, 0,      0,
            ↪  0],
            [0,      0,                0, 0,      0,               1, 0,      0,
            ↪  0],
            [0,      0,                0, 0,      0,               0, 1, deltaT,
            ↪  0.5*(deltaT**2)],
            [0,      0,                0, 0,      0,               0, 0,      1,
            ↪  deltaT],
            [0,      0,                0, 0,      0,               0, 0,      0,
            ↪  1]])

    # Process noise matrix
    Q = array([[(deltaT**4)/4, (deltaT**3)/2, (deltaT**2)/2, 0, 0, 0, 0, 0, 0],
               [(deltaT**3)/2,     deltaT**2,        deltaT, 0, 0, 0, 0, 0, 0],
               [(deltaT**2)/2,        deltaT,             1, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, (deltaT**4)/4, (deltaT**3)/2, (deltaT**2)/2, 0, 0, 0],
               [0, 0, 0, (deltaT**3)/2,     deltaT**2,        deltaT, 0, 0, 0],
               [0, 0, 0, (deltaT**2)/2,        deltaT,             1, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, (deltaT**4)/4, (deltaT**3)/2, (deltaT**2)/2],
               [0, 0, 0, 0, 0, 0, (deltaT**3)/2,     deltaT**2,        deltaT],
               [0, 0, 0, 0, 0, 0, (deltaT**2)/2,        deltaT,             1]])
    Q = Q * sigma_a**2

    # Observation matrix
    H = array([[1, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 1, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 1, 0, 0]])

    # Measurement covariance (uncertainty) matrix
    R = array([[sigma_m**2,          0,          0],
               [         0, sigma_m**2,          0],
               [         0,          0, sigma_m**2]])

    # Identity matrix the size and shape of F
    I = array(eye(F.shape[0]))

    # Predict the next state and covariance
    x = F @ x
    P = F @ P @ F.T + Q

    # Kalman gain
    K = P @ H.T @ linalg.inv(H @ P @ H.T + R)

    # State update
    x = x + K @ (z - H @ x)

    # Covariance update
    P = (I - K @ H) @ P @ (I - K @ H).T + K @ R @ K.T

    return x, P


# Function to predict the next step based on current Kalman state without updates
def predictKalman(x, P, deltaT, sigma_a):
    '''
        Inputs:
            x       -- Current state
            P       -- Current estimate uncertainty
            deltaT  -- Time difference since current state
            sigma_a -- Acceleration st dev

        Outputs:
            predx   -- Predicted next state
            predP   -- Predicted next estimate uncertainty

    '''
```

```
# State transition matrix
F = array([[1, deltaT, 0.5*(deltaT**2), 0,      0,               0, 0,      0,
↪  0],
           [0,      1,          deltaT, 0,      0,               0, 0,      0,
           ↪  0],
           [0,      0,               1, 0,      0,               0, 0,      0,
           ↪  0],
           [0,      0,               0, 1, deltaT, 0.5*(deltaT**2), 0,      0,
           ↪  0],
           [0,      0,               0, 0,      1,          deltaT, 0,      0,
           ↪  0],
           [0,      0,               0, 0,      0,               1, 0,      0,
           ↪  0],
           [0,      0,               0, 0,      0,               0, 1, deltaT,
           ↪  0.5*(deltaT**2)],
           [0,      0,               0, 0,      0,               0, 0,      1,
           ↪  deltaT],
           [0,      0,               0, 0,      0,               0, 0,      0,
           ↪  1]])

# Process noise matrix
Q = array([[(deltaT**4)/4, (deltaT**3)/2, (deltaT**2)/2, 0, 0, 0, 0, 0, 0],
           [(deltaT**3)/2,     deltaT**2,        deltaT, 0, 0, 0, 0, 0, 0],
           [(deltaT**2)/2,        deltaT,             1, 0, 0, 0, 0, 0, 0],
           [0, 0, 0, (deltaT**4)/4, (deltaT**3)/2, (deltaT**2)/2, 0, 0, 0],
           [0, 0, 0, (deltaT**3)/2,     deltaT**2,        deltaT, 0, 0, 0],
           [0, 0, 0, (deltaT**2)/2,        deltaT,             1, 0, 0, 0],
           [0, 0, 0, 0, 0, 0, (deltaT**4)/4, (deltaT**3)/2, (deltaT**2)/2],
           [0, 0, 0, 0, 0, 0, (deltaT**3)/2,     deltaT**2,        deltaT],
           [0, 0, 0, 0, 0, 0, (deltaT**2)/2,        deltaT,             1]])
Q = Q * sigma_a**2

# Predict next state and covariance
predx = F @ x
predP = F @ P @ F.T + Q

return predx, predP
```

```
# """
# baselineModels.R
#
# Notes:
#     The purpose of this module is to present baseline models, which are naive
#     models to "detect" defects. These models are to compare other more complex
#     techniques, since they should be better than the baseline.
#
# Use:
#     baselineDefectDetection(df)
#
#         Arguments:
#         df  -- DataFrame object containing flight data
#
#     Returns data structure with predictions and defect detection
#
# """

baselineDefectDetection <- function(df) {
    source(paste(getwd(), "/code/supportTools/logResultsSummary.R", sep=""))

    # First baseline model is predicting zero defects
    df$baselineDefectNone <- 0

    # Second baseline model is predicting all defects
    df$baselineDefectAll <- 1

    # Final baseline model is randomly predicting defects
    # Strategy is to use the probability of a defect from known data,
    # so let's calculate that...
    prob <- mean(df$rowDefect)

    rowCount <- nrow(df)

    # Initialize the RNG for repeatability across OS and R versions
    suppressWarnings(RNGkind(sample.kind = "Rounding"))
    set.seed(0)

    # Initialize baseline model to zeros and override to 1 later
    df$baselineDefectRandom <- 0

    # "Detect" defects... randomly
    for (i in 1:rowCount) {
        if (runif(1) < prob) {
            df[i,]$baselineDefectRandom <- 1
        }
    }

    # Log results in results summary
    logResultsSummary(df, "Baseline - Never Defect", "baselineDefectNone")
    logResultsSummary(df, "Baseline - Always Defect", "baselineDefectAll")
    logResultsSummary(df, "Baseline - Random", "baselineDefectRandom")

    print("Baseline models complete!")

    return(df)
}
```

```
# """
# polynomialRegression.R
#
# Notes:
#     The purpose of this module is to detect defects in the data using a
#     polynomial regression model. The errors from the model will determine what
#     is within the range of reasonability.
#
# Use:
#     polynomialRegression(df, lvl, multiplier, window)
#
#         Arguments:
#         df     -- DataFrame object containing flight data
#         lvl    -- (0, 1) Default 0.95. Prediction interval confidence. Tune
#                   higher for fewer false positives and lower for more
#                   sensitive detection of defects.
#         multiplier -- Default 1.0. Since confidence bands are very narrow,
#                       this multiplier will widen them
#         window -- Default 10. Moving window for data to feed into Regression model
#
#     Returns data structure with predictions and defect detection
#
#     TODO: Loop over a list of CI instead of redoing the model for each.
#
# """


polynomialRegression <- function(df, lvl=0.95, multiplier=1.0, window=10) {
    print(paste("Performing polynomial regression with ", lvl*100, "% CI and ", multiplier, "X
    ↪  multiplier and window ", window, "...", sep=""))
    print("One core in use, this is going to take a while...")

    source(paste(getwd(), "/code/supportTools/logResultsSummary.R", sep=""))

    rowCount <- nrow(df)

    # Initialize new columns in data frame
    df$polyXpred <- NA; df$polyXlower <- NA; df$polyXupper <- NA
    df$polyYpred <- NA; df$polyYlower <- NA; df$polyYupper <- NA
    df$polyZpred <- NA; df$polyZlower <- NA; df$polyZupper <- NA
    df$polyDefect <- NA

    # Setup progress bar
    pb = txtProgressBar(0, rowCount, style=3)

    for (i in 1:rowCount) {
        # Get flightID and time
        thisFlight <- df[i,]$flightID
        thisTime <- df[i,]$time

        # Filter dataframe to previous 10 entries
        tempdf <- tail(df[df$flightID==thisFlight & df$time<thisTime,], window)

        # Only perform regression defect detection if data is greater than window
        if (nrow(tempdf) >= window) {

            # Build polynomial regression models
            polyX <- lm(x ~ poly(time, 2), data=tempdf)
            polyY <- lm(y ~ poly(time, 2), data=tempdf)
            polyZ <- lm(z ~ poly(time, 2), data=tempdf)

            # Determine thresholds for defect detection
            # Since prediction bounds are very narrow, trying a multiplier
            xPred <- predict(polyX, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[1]
            xLower <- predict(polyX, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[2]
            xLower <- xPred - multiplier * (xPred - xLower)
```

```
            xUpper <- predict(polyX, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[3]
            xUpper <- xPred + multiplier * (xUpper - xPred)

            yPred <- predict(polyY, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[1]
            yLower <- predict(polyY, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[2]
            yLower <- yPred - multiplier * (yPred - yLower)
            yUpper <- predict(polyY, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[3]
            yUpper <- yPred + multiplier * (yUpper - yPred)

            zPred <- predict(polyZ, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[1]
            zLower <- predict(polyZ, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[2]
            zLower <- zPred - multiplier * (zPred - zLower)
            zUpper <- predict(polyZ, newdata=df[df$flightID==thisFlight & df$time==thisTime,],
            ↪  interval="prediction", level=lvl)[3]
            zUpper <- zPred + multiplier * (zUpper - zPred)

            # Save prediction and error bounds to dataframe
            df[df$flightID==thisFlight &
            ↪  df$time==thisTime,c("polyXpred","polyXlower","polyXupper")] <-
            ↪  c(xPred,xLower,xUpper)
            df[df$flightID==thisFlight &
            ↪  df$time==thisTime,c("polyYpred","polyYlower","polyYupper")] <-
            ↪  c(yPred,yLower,yUpper)
            df[df$flightID==thisFlight &
            ↪  df$time==thisTime,c("polyZpred","polyZlower","polyZupper")] <-
            ↪  c(zPred,zLower,zUpper)

            # Determine if prediction detects a defect
            conddf <- df[df$flightID==thisFlight & df$time==thisTime,] # dataframe for conditions

            cond <- expression( conddf$x>conddf$polyXupper |  conddf$x<conddf$polyXlower |
                                conddf$y>conddf$polyYupper |  conddf$y<conddf$polyYlower |
                                conddf$z>conddf$polyZupper |  conddf$z<conddf$polyZlower)

            ifelse(eval(cond), df[df$flightID==thisFlight & df$time==thisTime,]$polyDefect <- 1,
                   df[df$flightID==thisFlight & df$time==thisTime,]$polyDefect <- 0)

        } else {
            # Set this row to 0 if available data is too small, NA ruins stuff
            df[df$flightID==thisFlight & df$time==thisTime,]$polyDefect <- 0
        }

        # Advance progress bar
        setTxtProgressBar(pb,i)
    }

    # Log results in results summary
    logResultsSummary(df, paste("Second-Order Polynomial Regression - ", lvl*100, "% CI and ",
    ↪  multiplier, "X multiplier and window ", window, sep=""), "polyDefect")

    # Rename to include CI level
    names(df)[names(df) == 'polyDefect'] <- paste('polyDefect_', lvl, '_multiplier', multiplier,
    ↪  '_window', window, sep="")

    close(pb)
    print("Polynomial regression complete!")

    return(df)
}
```

```
# """
# arimaFunction.R
#
# Notes:
#     The purpose of this module is to detect defects in the data using an
#     autoregressive integrated moving average (ARIMA) model. The errors from
#     the model will determine what is within the range of reasonability.
#
# Use:
#     arimaFunction(df, lvl, multiplier)
#
#         Arguments:
#         df     -- DataFrame object containing flight data
#         lvl    -- (0, 1) Default 0.95. Prediction interval confidence. Tune
#                   higher for fewer false positives and lower for more
#                   sensitive detection of defects.
#         multiplier -- Default 1.0. Since ARIMA confidence bands are very narrow,
#                       this multiplier will widen them
#         p      -- (Required) Autoregressive (AR) order
#         d      -- (Required) Stationarity correction (I)
#         q      -- (Required) Moving Average (AR) order
#         window -- Default 10. Moving window for data to feed into ARIMA model
#
#     Returns data structure with predictions and defect detection
#
#     TODO: Loop over a list of CI instead of redoing the model for each.
#
# """

arimaFunction <- function(df, lvl=0.95, multiplier=1.0, p, d, q, window) {

    print(paste("Performing ARIMA(",p,",",d,",",q,") with ", lvl*100, "% CI and ", multiplier, "X
    ↪  multiplier and window ", window, "...", sep=""))
    print("One core in use, this is going to take a while...")

    library(forecast)

    source(paste(getwd(), "/code/supportTools/logResultsSummary.R", sep=""))

    rowCount <- nrow(df)

    # Initialize new columns in data frame
    df$arimaXpred <- NA; df$arimaXlower <- NA; df$arimaXupper <- NA
    df$arimaYpred <- NA; df$arimaYlower <- NA; df$arimaYupper <- NA
    df$arimaZpred <- NA; df$arimaZlower <- NA; df$arimaZupper <- NA
    df$arimaDefect <- NA

    # Setup progress bar
    pb = txtProgressBar(0, rowCount, style=3)

    for (i in 1:rowCount) {
        # Get flightID and time
        thisFlight <- df[i,]$flightID
        thisTime <- df[i,]$time

        # Filter dataframe to previous 10 entries
        tempdf <- tail(df[df$flightID==thisFlight & df$time<thisTime,], window)

        # Only perform defect detection if data is greater than window
        if (nrow(tempdf) >= window) {

            # Build ARIMA models
            arimaX <- arima(tempdf$x, order=c(p,d,q), method="CSS")
            arimaY <- arima(tempdf$y, order=c(p,d,q), method="CSS")
            arimaZ <- arima(tempdf$z, order=c(p,d,q), method="CSS")


            # Determine thresholds for defect detection
```

```
    # Since ARIMA prediction bounds are very narrow, trying a multiplier

    # First, some R trickery to catch a rare issue with infinite bounds
    getLower <- function(arimaModel, lvl, coord) {
        if (is.na(forecast(arimaModel, h=1, level=lvl)$lower[1])) {
            return(coord-0.0000001)
        } else {
            return(forecast(arimaModel, h=1, level=lvl)$lower[1])
        }
    }
    getUpper <- function(arimaModel, lvl, coord) {
        if (is.na(forecast(arimaModel, h=1, level=lvl)$upper[1])) {
            return(coord+0.0000001)
        } else {
            return(forecast(arimaModel, h=1, level=lvl)$upper[1])
        }
    }


    xPred <- forecast(arimaX, h=1, level=lvl)$mean[1]
    xLower <- getLower(arimaX, lvl, xPred)
    xLower <- xPred - multiplier * (xPred - xLower)
    xUpper <- getUpper(arimaX, lvl, xPred)
    xUpper <- xPred + multiplier * (xUpper - xPred)

    yPred <- forecast(arimaY, h=1, level=lvl)$mean[1]
    yLower <- getLower(arimaY, lvl, yPred)
    yLower <- yPred - multiplier * (yPred - yLower)
    yUpper <- getUpper(arimaY, lvl, yPred)
    yUpper <- yPred + multiplier * (yUpper - yPred)

    zPred <- forecast(arimaZ, h=1, level=lvl)$mean[1]
    zLower <- getLower(arimaZ, lvl, zPred)
    zLower <- zPred - multiplier * (zPred - zLower)
    zUpper <- getUpper(arimaZ, lvl, zPred)
    zUpper <- zPred + multiplier * (zUpper - zPred)

    # Determine if prediction detects a defect
    condX <- df[df$flightID==thisFlight & df$time==thisTime,]$x < xLower |
        df[df$flightID==thisFlight & df$time==thisTime,]$x > xUpper
    condY <- df[df$flightID==thisFlight & df$time==thisTime,]$y < yLower |
        df[df$flightID==thisFlight & df$time==thisTime,]$y > yUpper
    condZ <- df[df$flightID==thisFlight & df$time==thisTime,]$z < zLower |
        df[df$flightID==thisFlight & df$time==thisTime,]$z > zUpper

    cond <- expression(condX | condY | condZ)

    ifelse(eval(cond), df[df$flightID==thisFlight & df$time==thisTime,]$arimaDefect <- 1,
           df[df$flightID==thisFlight & df$time==thisTime,]$arimaDefect <- 0)

    # Save prediction and error bounds to dataframe
    df[df$flightID==thisFlight &
    ↪  df$time==thisTime,c("arimaXpred","arimaXlower","arimaXupper")] <-
        c(xPred, xLower, xUpper)
    df[df$flightID==thisFlight &
    ↪  df$time==thisTime,c("arimaYpred","arimaYlower","arimaYupper")] <-
        c(yPred, yLower, yUpper)
    df[df$flightID==thisFlight &
    ↪  df$time==thisTime,c("arimaZpred","arimaZlower","arimaZupper")] <-
        c(zPred, zLower, zUpper)

} else {
    # Set this row to 0 if available data is too small, NA ruins stuff
    df[df$flightID==thisFlight & df$time==thisTime,]$arimaDefect <- 0
}

# Advance progress bar
setTxtProgressBar(pb,i)
```

```
    }

    # Log results in results summary
    logResultsSummary(df, paste("ARIMA(",p,",",d,",",q,") - ", lvl*100, "% CI and ", multiplier,
    ↪  "x and window ", window, sep=""), "arimaDefect")

    # Rename to include CI level
    names(df)[names(df) == 'arimaDefect'] <- paste("arima(",p,",",d,",",q,")Defect_",
    ↪  multiplier,"X and window ", window, sep="")

    close(pb)
    print("ARIMA complete!")

    return(df)
}
```

```
# """
# neuralNet.R
#
# Notes:
#     The purpose of this module is to detect defects in the data using a
#     Neural Network model. The errors from the model will determine what
#     is within the range of reasonability.
#
# Use:
#     neuralNet(dfTrain, dfVal, dfTest, multiplier, hiddenSize, delayValue, testRun)
#
#         Arguments:
#         df         -- DataFrame object containing flight data (one each for
#                       Training, Validation, and Test)
#         multiplier -- Array. Default 1.0. Since confidence bands are very narrow,
#                       this multiplier will widen them
#         hiddenSize -- Default 5. Number of units in the hidden layer
#         delay      -- [0,1]. Delay value for nnet
#         testRun    -- Binary. Indicate a test run (0 for faster validation runs)
#
#     Returns data structure with predictions and defect detection
#
#     TODO: Loop over a list of CI instead of redoing the model for each.
#
# """
neuralNet <- function(dfTrain, dfVal, dfTest, multiplier=c(1.0), hiddenSize=5, delayValue=0,
↪  testRun=0) {
    library(nnet)
    library(dplyr)
    library(boot)
    source(paste(getwd(), "/code/supportTools/logResultsSummary.R", sep=""))

    print(paste("Beginning neural network model with [", paste(multiplier,collapse=" "), "]
    ↪  multiplier and hidden size ", hiddenSize, " and delay ", delayValue, "...", sep=""))
    if (testRun) {print("INFORMATION: This is a test run, which takes longer. Set testRun=0 for
    ↪  Validation runs.")}

    # Set some values
    runs = 100
    boots = 50

    ############## X Coordinate ##############
    # Add lag to train data X
    dfTrain <- dfTrain %>%
        group_by(flightID) %>%
        mutate(xLag1 = lag(xClean, n = 1), xLag2 = lag(xClean, n = 2), xLag3 = lag(xClean, n =
        ↪  3), xLag4 = lag(xClean, n = 4))
    dfTrain$x <- dfTrain$xClean # Use only the clean data to train model

    # Add lag to validation data X
    dfVal <- dfVal %>%
        group_by(flightID) %>%
        mutate(xLag1 = lag(x, n = 1), xLag2 = lag(x, n = 2), xLag3 = lag(x, n = 3), xLag4 =
        ↪  lag(x, n = 4))

    # Add lag to training data X
    dfTest <- dfTest %>%
        group_by(flightID) %>%
        mutate(xLag1 = lag(x, n = 1), xLag2 = lag(x, n = 2), xLag3 = lag(x, n = 3), xLag4 =
        ↪  lag(x, n = 4))


    ############## Y Coordinate ##############
    # Add lag to train data Y
    dfTrain <- dfTrain %>%
        group_by(flightID) %>%
        mutate(yLag1 = lag(yClean, n = 1), yLag2 = lag(yClean, n = 2), yLag3 = lag(yClean, n =
        ↪  3), yLag4 = lag(yClean, n = 4))
```

```
dfTrain$y <- dfTrain$yClean # Use only the clean data to train model

# Add lag to validation data Y
dfVal <- dfVal %>%
    group_by(flightID) %>%
    mutate(yLag1 = lag(y, n = 1), yLag2 = lag(y, n = 2), yLag3 = lag(y, n = 3), yLag4 =
    ↪  lag(y, n = 4))

# Add lag to training data Y
dfTest <- dfTest %>%
    group_by(flightID) %>%
    mutate(yLag1 = lag(y, n = 1), yLag2 = lag(y, n = 2), yLag3 = lag(y, n = 3), yLag4 =
    ↪  lag(y, n = 4))


############## Z Coordinate ##############
# Add lag to train data Z
dfTrain <- dfTrain %>%
    group_by(flightID) %>%
    mutate(zLag1 = lag(zClean, n = 1), zLag2 = lag(zClean, n = 2), zLag3 = lag(zClean, n =
    ↪  3), zLag4 = lag(zClean, n = 4))
dfTrain$z <- dfTrain$zClean # Use only the clean data to train model

# Add lag to validation data Z
dfVal <- dfVal %>%
    group_by(flightID) %>%
    mutate(zLag1 = lag(z, n = 1), zLag2 = lag(z, n = 2), zLag3 = lag(z, n = 3), zLag4 =
    ↪  lag(z, n = 4))

# Add lag to training data Z
dfTest <- dfTest %>%
    group_by(flightID) %>%
    mutate(zLag1 = lag(z, n = 1), zLag2 = lag(z, n = 2), zLag3 = lag(z, n = 3), zLag4 =
    ↪  lag(z, n = 4))


# Define the formula for the model
formulaX <- x ~ xLag1 + xLag2 + xLag3 + xLag4
formulaY <- y ~ yLag1 + yLag2 + yLag3 + yLag4
formulaZ <- z ~ zLag1 + zLag2 + zLag3 + zLag4

# Initialize predictions
predictionsValX <- data.frame(matrix(NA, nrow = nrow(dfVal[!is.na(dfVal$xLag4),]), ncol = 1))
predictionsValY <- data.frame(matrix(NA, nrow = nrow(dfVal[!is.na(dfVal$yLag4),]), ncol = 1))
predictionsValZ <- data.frame(matrix(NA, nrow = nrow(dfVal[!is.na(dfVal$zLag4),]), ncol = 1))

if (testRun) {
    predictionsTestX <- data.frame(matrix(NA, nrow = nrow(dfTest[!is.na(dfTest$xLag4),]),
    ↪  ncol = 1))
    predictionsTestY <- data.frame(matrix(NA, nrow = nrow(dfTest[!is.na(dfTest$yLag4),]),
    ↪  ncol = 1))
    predictionsTestZ <- data.frame(matrix(NA, nrow = nrow(dfTest[!is.na(dfTest$zLag4),]),
    ↪  ncol = 1))
}

# Setup progress bar
print("Building models...")
pb = txtProgressBar(0, runs, style=3)

# Create many models to do bootstrapping
for (i in 1:runs) {
    # Set seed, for repeatability
    #set.seed(i+seed)

    # Train the MLP model
    modelX <- nnet(formulaX, data = dfTrain[!is.na(dfTrain$xLag4),], size=hiddenSize,
    ↪  linout=TRUE, skip=TRUE, MaxNWts=10000, trace=FALSE, maxit=1000, delay=delayValue)
```

```
    modelY <- nnet(formulaY, data = dfTrain[!is.na(dfTrain$yLag4),], size=hiddenSize,
    ↪  linout=TRUE, skip=TRUE, MaxNWts=10000, trace=FALSE, maxit=1000, delay=delayValue)
    modelZ <- nnet(formulaZ, data = dfTrain[!is.na(dfTrain$zLag4),], size=hiddenSize,
    ↪  linout=TRUE, skip=TRUE, MaxNWts=10000, trace=FALSE, maxit=1000, delay=delayValue)

    # Make predictions on the validation set
    predictionsValX <- data.frame(predictionsValX, placeholder=predict(modelX,
    ↪  dfVal[!is.na(dfVal$xLag4),]))
    names(predictionsValX)[names(predictionsValX) == "placeholder"] <- paste("Run",i,sep="")
    predictionsValY <- data.frame(predictionsValY, placeholder=predict(modelY,
    ↪  dfVal[!is.na(dfVal$yLag4),]))
    names(predictionsValY)[names(predictionsValY) == "placeholder"] <- paste("Run",i,sep="")
    predictionsValZ <- data.frame(predictionsValZ, placeholder=predict(modelZ,
    ↪  dfVal[!is.na(dfVal$zLag4),]))
    names(predictionsValZ)[names(predictionsValZ) == "placeholder"] <- paste("Run",i,sep="")

    # Make predictions on the test set
    if (testRun) {
        predictionsTestX <- data.frame(predictionsTestX, placeholder=predict(modelX,
        ↪  dfTest[!is.na(dfTest$xLag4),]))
        names(predictionsTestX)[names(predictionsTestX) == "placeholder"] <-
        ↪  paste("Run",i,sep="")
        predictionsTestY <- data.frame(predictionsTestY, placeholder=predict(modelY,
        ↪  dfTest[!is.na(dfTest$yLag4),]))
        names(predictionsTestY)[names(predictionsTestY) == "placeholder"] <-
        ↪  paste("Run",i,sep="")
        predictionsTestZ <- data.frame(predictionsTestZ, placeholder=predict(modelZ,
        ↪  dfTest[!is.na(dfTest$zLag4),]))
        names(predictionsTestZ)[names(predictionsTestZ) == "placeholder"] <-
        ↪  paste("Run",i,sep="")
    }

    # Advance progress bar
    setTxtProgressBar(pb,i)
}

close(pb)
predictionsValX$matrix.NA..nrow...nrow.dfVal..is.na.dfVal.xLag4.......ncol...1. <- NULL
predictionsValY$matrix.NA..nrow...nrow.dfVal..is.na.dfVal.yLag4.......ncol...1. <- NULL
predictionsValZ$matrix.NA..nrow...nrow.dfVal..is.na.dfVal.zLag4.......ncol...1. <- NULL
if (testRun) {
    predictionsTestX$matrix.NA..nrow...nrow.dfTest..is.na.dfTest.xLag4.......ncol...1. <-
    ↪  NULL
    predictionsTestY$matrix.NA..nrow...nrow.dfTest..is.na.dfTest.yLag4.......ncol...1. <-
    ↪  NULL
    predictionsTestZ$matrix.NA..nrow...nrow.dfTest..is.na.dfTest.zLag4.......ncol...1. <-
    ↪  NULL
}


# Store predictions in a CSV for each coordinate
print("Saving data...")
write.csv(predictionsValX, paste(getwd(), "/dataValidation/predictionsX.csv", sep=""),
↪  row.names=FALSE)
write.csv(predictionsValY, paste(getwd(), "/dataValidation/predictionsY.csv", sep=""),
↪  row.names=FALSE)
write.csv(predictionsValZ, paste(getwd(), "/dataValidation/predictionsZ.csv", sep=""),
↪  row.names=FALSE)
if (testRun) {
    write.csv(predictionsTestX, paste(getwd(), "/dataTest/predictionsX.csv", sep=""),
    ↪  row.names=FALSE)
    write.csv(predictionsTestY, paste(getwd(), "/dataTest/predictionsY.csv", sep=""),
    ↪  row.names=FALSE)
    write.csv(predictionsTestZ, paste(getwd(), "/dataTest/predictionsZ.csv", sep=""),
    ↪  row.names=FALSE)
}

# Bootstrap the confidence intervals
```

```
bootFun <- function(df,idx)
{
    estimate <- mean(df[idx,])
}

print("Bootstrapping Validation confidence intervals...")
pb = txtProgressBar(0, nrow(predictionsValX), style=3)

# Initialize data format
confIntValX <- data.frame(matrix(NA, nrow = nrow(predictionsValX), ncol = 3)) %>% rename(Mean
↪  = X1, Lower = X2, Upper = X3)
confIntValY <- data.frame(matrix(NA, nrow = nrow(predictionsValY), ncol = 3)) %>% rename(Mean
↪  = X1, Lower = X2, Upper = X3)
confIntValZ <- data.frame(matrix(NA, nrow = nrow(predictionsValZ), ncol = 3)) %>% rename(Mean
↪  = X1, Lower = X2, Upper = X3)

# Bootstrap confidence intervals for Validation data
for (i in 1:nrow(predictionsValX)) {
    # Set seed, for repeatability
    #set.seed(i+seed)

    bootstrapX <- boot(t(predictionsValX[i,]), bootFun, R = boots)
    bootOutX <- boot.ci(boot.out = bootstrapX, type = "norm")
    bootstrapY <- boot(t(predictionsValY[i,]), bootFun, R = boots)
    bootOutY <- boot.ci(boot.out = bootstrapY, type = "norm")
    bootstrapZ <- boot(t(predictionsValZ[i,]), bootFun, R = boots)
    bootOutZ <- boot.ci(boot.out = bootstrapZ, type = "norm")

    # Record confidence intervals with multiplier
    confIntValX$Mean[i] <- bootOutX$t0
    confIntValX$Lower[i] <- bootOutX$normal[1,2]
    confIntValX$Upper[i] <- bootOutX$normal[1,3]
    confIntValY$Mean[i] <- bootOutY$t0
    confIntValY$Lower[i] <- bootOutY$normal[1,2]
    confIntValY$Upper[i] <- bootOutY$normal[1,3]
    confIntValZ$Mean[i] <- bootOutZ$t0
    confIntValZ$Lower[i] <- bootOutZ$normal[1,2]
    confIntValZ$Upper[i] <- bootOutZ$normal[1,3]

    # Advance progress bar
    setTxtProgressBar(pb,i)
}
close(pb)

# Store confidence intervals in dfVal
dfVal$nnetXpred <- NA; dfVal$nnetXlower <- NA; dfVal$nnetXupper <- NA
dfVal$nnetYpred <- NA; dfVal$nnetYlower <- NA; dfVal$nnetYupper <- NA
dfVal$nnetZpred <- NA; dfVal$nnetZlower <- NA; dfVal$nnetZupper <- NA
dfVal$nnetDefect <- 0

print("Applying array of multipliers...")
pb = txtProgressBar(0, length(multiplier), style=3)

for (mult in multiplier) {
    dfVal[!is.na(dfVal$xLag4),]$nnetXpred <- confIntValX$Mean
    dfVal[!is.na(dfVal$xLag4),]$nnetXlower <- confIntValX$Mean - mult * (confIntValX$Mean -
    ↪  confIntValX$Lower)
    dfVal[!is.na(dfVal$xLag4),]$nnetXupper <- confIntValX$Mean + mult * (confIntValX$Upper-
    ↪  confIntValX$Mean)
    dfVal[!is.na(dfVal$yLag4),]$nnetYpred <- confIntValY$Mean
    dfVal[!is.na(dfVal$yLag4),]$nnetYlower <- confIntValY$Mean - mult * (confIntValY$Mean -
    ↪  confIntValY$Lower)
    dfVal[!is.na(dfVal$yLag4),]$nnetYupper <- confIntValY$Mean + mult * (confIntValY$Upper-
    ↪  confIntValY$Mean)
    dfVal[!is.na(dfVal$zLag4),]$nnetZpred <- confIntValZ$Mean
    dfVal[!is.na(dfVal$zLag4),]$nnetZlower <- confIntValZ$Mean - mult * (confIntValZ$Mean -
    ↪  confIntValZ$Lower)
```

```
    dfVal[!is.na(dfVal$zLag4),]$nnetZupper <- confIntValZ$Mean + mult * (confIntValZ$Upper-
    ↪  confIntValZ$Mean)

    # Determine if prediction detects a defect
    cond <- expression( dfVal$x>dfVal$nnetXupper |  dfVal$x<dfVal$nnetXlower |
                        dfVal$y>dfVal$nnetYupper |  dfVal$y<dfVal$nnetYlower |
                        dfVal$z>dfVal$nnetZupper |  dfVal$z<dfVal$nnetZlower)

    dfVal$nnetDefect <- ifelse(eval(cond), dfVal$nnetDefect <- 1, dfVal$nnetDefect <- 0)
    dfVal$nnetDefect <- dfVal$nnetDefect %>% replace(is.na(.), 0) #Replace NA with 0

    # Remove unnecessary columns and save to file
    write.csv(dfVal, paste(getwd(), "/dataValidation/dataValidation.csv", sep=""),
    ↪  row.names=FALSE)

    # Log results
    logResultsSummary(dfVal, paste("Neural Net - ", mult, "X multiplier and hidden size ",
    ↪  hiddenSize, " and delay ", delayValue, sep=""), "nnetDefect", "validation")

    # Advance progress bar
    setTxtProgressBar(pb,match(mult,multiplier))
}
close(pb)

if (testRun) {
    print("Bootstrapping Test confidence intervals...")
    pb = txtProgressBar(0, nrow(predictionsTestX), style=3)

    # Initialize data format
    confIntTestX <- data.frame(matrix(NA, nrow = nrow(predictionsTestX), ncol = 3)) %>%
    ↪  rename(Mean = X1, Lower = X2, Upper = X3)
    confIntTestY <- data.frame(matrix(NA, nrow = nrow(predictionsTestY), ncol = 3)) %>%
    ↪  rename(Mean = X1, Lower = X2, Upper = X3)
    confIntTestZ <- data.frame(matrix(NA, nrow = nrow(predictionsTestZ), ncol = 3)) %>%
    ↪  rename(Mean = X1, Lower = X2, Upper = X3)

    # Bootstrap confidence intervals for Test data
    for (i in 1:nrow(predictionsTestX)) {
        # Set seed, for repeatability
        #set.seed(i+seed)

        bootstrapX <- boot(t(predictionsTestX[i,]), bootFun, R = boots)
        bootOutX <- boot.ci(boot.out = bootstrapX, type = "norm")
        bootstrapY <- boot(t(predictionsTestY[i,]), bootFun, R = boots)
        bootOutY <- boot.ci(boot.out = bootstrapY, type = "norm")
        bootstrapZ <- boot(t(predictionsTestZ[i,]), bootFun, R = boots)
        bootOutZ <- boot.ci(boot.out = bootstrapZ, type = "norm")

        # Record confidence intervals with multiplier
        confIntTestX$Mean[i] <- bootOutX$t0
        confIntTestX$Lower[i] <- bootOutX$normal[1,2]
        confIntTestX$Upper[i] <- bootOutX$normal[1,3]
        confIntTestY$Mean[i] <- bootOutY$t0
        confIntTestY$Lower[i] <- bootOutY$normal[1,2]
        confIntTestY$Upper[i] <- bootOutY$normal[1,3]
        confIntTestZ$Mean[i] <- bootOutZ$t0
        confIntTestZ$Lower[i] <- bootOutZ$normal[1,2]
        confIntTestZ$Upper[i] <- bootOutZ$normal[1,3]

        # Advance progress bar
        setTxtProgressBar(pb,i)
    }
    close(pb)

    # Store confidence intervals in dfTest
    dfTest$nnetXpred <- NA; dfTest$nnetXlower <- NA; dfTest$nnetXupper <- NA
    dfTest$nnetYpred <- NA; dfTest$nnetYlower <- NA; dfTest$nnetYupper <- NA
    dfTest$nnetZpred <- NA; dfTest$nnetZlower <- NA; dfTest$nnetZupper <- NA
```

```
        dfTest$nnetDefect <- 0

        print("Applying array of multipliers...")
        pb = txtProgressBar(0, length(multiplier), style=3)

        for (mult in multiplier) {
            dfTest[!is.na(dfTest$xLag4),]$nnetXpred <- confIntTestX$Mean
            dfTest[!is.na(dfTest$xLag4),]$nnetXlower <- confIntTestX$Mean - mult *
            ↪  (confIntTestX$Mean - confIntTestX$Lower)
            dfTest[!is.na(dfTest$xLag4),]$nnetXupper <- confIntTestX$Mean + mult *
            ↪  (confIntTestX$Upper- confIntTestX$Mean)
            dfTest[!is.na(dfTest$yLag4),]$nnetYpred <- confIntTestY$Mean
            dfTest[!is.na(dfTest$yLag4),]$nnetYlower <- confIntTestY$Mean - mult *
            ↪  (confIntTestY$Mean - confIntTestY$Lower)
            dfTest[!is.na(dfTest$yLag4),]$nnetYupper <- confIntTestY$Mean + mult *
            ↪  (confIntTestY$Upper- confIntTestY$Mean)
            dfTest[!is.na(dfTest$zLag4),]$nnetZpred <- confIntTestZ$Mean
            dfTest[!is.na(dfTest$zLag4),]$nnetZlower <- confIntTestZ$Mean - mult *
            ↪  (confIntTestZ$Mean - confIntTestZ$Lower)
            dfTest[!is.na(dfTest$zLag4),]$nnetZupper <- confIntTestZ$Mean + mult *
            ↪  (confIntTestZ$Upper- confIntTestZ$Mean)

            # Determine if prediction detects a defect
            cond <- expression( dfTest$x>dfTest$nnetXupper |  dfTest$x<dfTest$nnetXlower |
                                dfTest$y>dfTest$nnetYupper |  dfTest$y<dfTest$nnetYlower |
                                dfTest$z>dfTest$nnetZupper |  dfTest$z<dfTest$nnetZlower)

            dfTest$nnetDefect <- ifelse(eval(cond), dfTest$nnetDefect <- 1, dfTest$nnetDefect <-
            ↪  0)
            dfTest$nnetDefect <- dfTest$nnetDefect %>% replace(is.na(.), 0) #Replace NA with 0

            # Remove unnecessary columns and save to file
            write.csv(dfTest, paste(getwd(), "/dataTest/dataTest.csv", sep=""), row.names=FALSE)

            # Log results
            logResultsSummary(dfVal, paste("Neural Net - ", mult, "X multiplier and hidden size
            ↪  ", hiddenSize, " and delay ", delayValue, sep=""), "nnetDefect", "test")

            # Advance progress bar
            setTxtProgressBar(pb,match(mult,multiplier))
        }
        close(pb)
    }

    print("Neural network model complete!")
}
```

```
# """
# metaAlgorithm.R
#
# Notes:
#     The purpose of this module is to detect defects in the data using a
#     meta algorithm combining the results of the other three techniques.
#
# Use:
#     metaAlgorithm(dfTrain, dfVal, dfTest, hiddenSize, delayValue)
#
#         Arguments:
#         df         -- DataFrame object containing flight data (one each for
#                       Validation, and Test)
#         hiddenSize -- Default 5. Number of units in the hidden layer
#         delay      -- [0,1]. Delay value for nnet
#         testRun    -- Binary. Indicate a test run (0 for faster validation runs)
#
#     Returns data structure with predictions and defect detection
#
#     TODO: Loop over a list of CI instead of redoing the model for each.
#
# """
metaAlgorithm <- function(dfTrain, dfVal, dfTest, hiddenSize=5, delayValue=0, testRun=0) {
    library(nnet)
    library(dplyr)
    source(paste(getwd(), "/code/supportTools/logResultsSummary.R", sep=""))

    print(paste("Beginning meta learning model with hidden size ", hiddenSize, " and delay ",
    ↪  delayValue, "...", sep=""))
    if (testRun) {print("This run will use Test data.")}

    # Create variable with proporition of time elapsed in flight from 0 to 1
    dfTrain <- merge(dfTrain, aggregate(time~flightID, dfTrain, FUN=max), all.dfTrain=TRUE,
    ↪  by="flightID")
    names(dfTrain)[names(dfTrain)=='time.y'] <- 'maxTime'
    names(dfTrain)[names(dfTrain)=='time.x'] <- 'time'
    dfTrain$timeProp <- dfTrain$time / dfTrain$maxTime

    dfVal <- merge(dfVal, aggregate(time~flightID, dfVal, FUN=max), all.dfVal=TRUE,
    ↪  by="flightID")
    names(dfVal)[names(dfVal)=='time.y'] <- 'maxTime'
    names(dfVal)[names(dfVal)=='time.x'] <- 'time'
    dfVal$timeProp <- dfVal$time / dfVal$maxTime

    if (testRun) {
        dfTest <- merge(dfTest, aggregate(time~flightID, dfTest, FUN=max), all.dfTest=TRUE,
        ↪  by="flightID")
        names(dfTest)[names(dfTest)=='time.y'] <- 'maxTime'
        names(dfTest)[names(dfTest)=='time.x'] <- 'time'
        dfTest$timeProp <- dfTest$time / dfTest$maxTime
    }

    # Define the formula for the model
    formula <- rowDefect ~ timeProp + polyDefect_0.99_multiplier20_window6 +
    ↪  arima.2.2.0.Defect_600X.and.window.8 + nnetDefect

    # Build the model against Training data
    model <- nnet(formula, data = dfTrain, size=hiddenSize, linout=TRUE, skip=TRUE,
    ↪  MaxNWts=10000, trace=FALSE, maxit=1000, delay=delayValue)

    # Predict defects for Validation and Test
    dfVal$metaAlg <- round(predict(model, dfVal))
    if (testRun) { dfTest$metaAlg <- round(predict(model, dfTest)) }

    # Log results
    logResultsSummary(dfVal, paste("Meta Algorithm - hidden size ", hiddenSize, " and delay ",
    ↪  delayValue, sep=""), "metaAlg", "validation")
```

```
    if (testRun) {logResultsSummary(dfTest, paste("Meta Algorithm - hidden size ", hiddenSize, "
    ↪  and delay ", delayValue, sep=""), "metaAlg", "test")}

    # Save data
    print("Saving data...")
    write.csv(dfVal, paste(getwd(), "/dataValidation/dataValidation.csv", sep=""),
    ↪  row.names=FALSE)
    if (testRun) {write.csv(dfTest, paste(getwd(), "/dataTest/dataTest.csv", sep=""),
    ↪  row.names=FALSE)}

    print("Meta learning model complete!")
}
```

```
"""
repeatableRandom.py

Notes:
    The purpose of this function is to produce a repeatable random number stream
    across the entire project that is system and architecture agnostic. NumPy
    provides such a pseudo random number generator, but it is not guaranteed to
    be repeatable across NumPy versions. For this reason, Numpy will be forced
    to a specific version

    It is important to realize that changing certain portions of code that call
    this function may produce different results going forward.

Use:
    repRand()

    Returns a random number between 0 and 1

"""

from pkg_resources import require
require("numpy==1.24.3")  # modified to use specific numpy
from numpy.random import default_rng

# Set the random number seed and do not change it to keep results repeatable
seed = 0

# Create random number generator with seed
rg = default_rng(seed)

# Define the functions, so that it can be used elsewhere
def repRand():
    return rg.random()

def repRandInt(low, high):
    return rg.integers(low, high, endpoint=True)

def repRandDec(low, high):
    return rg.random() * (high - low) + low

def choice(count=1, size=1, replace=False):
    return rg.choice(count, size, replace)


# Reset the generator
def repRandReset(seed=0):
    global rg
    del rg
    rg = default_rng(seed)
```

```
# """
# logResultsSummary.R
#
# Notes:
#     The purpose of this module is to log and calculate the summary statistics
#     from the defect detection results. The summary is saved to a CSV file.
#
# Use:
#     logResultsSummary(df, technique, column)
#
#         Arguments:
#         df        -- Dataframe containing defect data
#         technique -- Name of the technique used
#         column    -- Column to evaluate summary (ex. "defectDetectionName")
#         dataType  -- Optional. Set where to save the data
#
#     Saves a CSV with results summary
#
#     logResults(df)
#
#         Arguments:
#         df        -- Dataframe containing defect data
#
#     Does all the logging for all the techniques so far, manually updated
#
# """



logResultsSummary <- function(df, Technique, column, dataType="train") {
    # Big TODO here, need to add a parameter to other files to pass this in. Just
    # use it as a switch for now.
    location <- "/data"
    #location <- "/dataTest"

    # Auto switch, but needs to be implemented in polynomial and ARIMA
    if(dataType == "validation") {
        location <- "/dataValidation"
    } else if (dataType == "test") {
        location <- "/dataTest"
    }

    # Read current file, or create it if it does not exist
    if (file.exists(paste(getwd(), location, "/resultsSummary.csv", sep=""))) {
        csv <- read.csv(paste(getwd(), location, "/resultsSummary.csv", sep=""))
    } else {
        csv <- list("Technique", "TrueNegative", "TruePositiveLow",
                    "TruePositiveMed", "TruePositiveHigh", "FalsePositive",
                    "FalseNegativeLow", "FalseNegativeMed", "FalseNegativeHigh",
                    "Accuracy", "Specificity",
                    "Recall", "Recall_L", "Recall_U", "Precision", "Precision_L", "Precision_U",
                    "NPV", "F1", "F1_L", "F1_U", "F0.5", "F2")
        write.table(csv, paste(getwd(), location, "/resultsSummary.csv", sep=""), sep=",",
        ↪  col.names=FALSE, row.names=FALSE)
        csv <- read.csv(paste(getwd(), location, "/resultsSummary.csv", sep=""))
    }

    # Check if current technique already exists. If so, remove it.
    csv <- csv[csv$Technique != Technique, ]


    # Calculate the results of defect detection
    TrueNegative <- nrow(df[which(df$rowDefect==0 & df[,column]==0), ])
    TruePositiveLow <- nrow(df[which(df$rowDefect==1 & df[,column]==1 &
    ↪  df$defectSeverity=="Low"), ])
    TruePositiveMed <- nrow(df[which(df$rowDefect==1 & df[,column]==1 &
    ↪  df$defectSeverity=="Med"), ])
```

```
TruePositiveHigh <- nrow(df[which(df$rowDefect==1 & df[,column]==1 &
↪  df$defectSeverity=="High"), ])
FalsePositive <- nrow(df[which(df$rowDefect==0 & df[,column]==1), ])
FalseNegativeLow <- nrow(df[which(df$rowDefect==1 & df[,column]==0 &
↪  df$defectSeverity=="Low"), ])
FalseNegativeMed <- nrow(df[which(df$rowDefect==1 & df[,column]==0 &
↪  df$defectSeverity=="Med"), ])
FalseNegativeHigh <- nrow(df[which(df$rowDefect==1 & df[,column]==0 &
↪  df$defectSeverity=="High"), ])

# Group results into false/true positive/negative
TruePositive  <- TruePositiveLow + TruePositiveMed + TruePositiveHigh
FalseNegative <- FalseNegativeLow + FalseNegativeMed + FalseNegativeHigh

# Calculate metrics
Accuracy    <- (TruePositive + TrueNegative) /
               (TruePositive + TrueNegative + FalsePositive + FalseNegative)

Specificity <- TrueNegative / (TrueNegative + FalsePositive)

Recall      <- TruePositive / (TruePositive + FalseNegative)

Precision   <- TruePositive / (TruePositive + FalsePositive)

NPV         <- TrueNegative / (TrueNegative + FalseNegative)

# Setup ThresholdROC
library(ThresholdROC)
tab <- as.table(matrix(c(TruePositive, FalsePositive, FalseNegative, TrueNegative), ncol=2,
↪  byrow=TRUE))
result <- tryCatch({result <- diagnostic(tab)}, error=function(e){result <- as.table(0)})

# If able, calculate the lower and upper bounds
if (dim(result)[1] == 1) {
    RecallL <- NA
    RecallU <- NA
    PrecL  <- NA
    PrecU  <- NA
} else {
    RecallL <- result["Sensitivity","Low.lim(95%)"]
    RecallU <- result["Sensitivity","Up.lim(95%)"]
    PrecL  <- result["Pos.Pred.Val.","Low.lim(95%)"]
    PrecU  <- result["Pos.Pred.Val.","Up.lim(95%)"]
}

Fscore_beta <- 1
F1 <- (1+Fscore_beta^2)*TruePositive /
          ((1+Fscore_beta^2)*TruePositive + Fscore_beta^2*FalseNegative + FalsePositive)
F1L <- NA #upper and lower bounds found in FmeasureBootstrapping.R manually for now
F1U <- NA

Fscore_beta <- 0.5
Fhalf <- (1+Fscore_beta^2)*TruePositive /
    ((1+Fscore_beta^2)*TruePositive + Fscore_beta^2*FalseNegative + FalsePositive)

Fscore_beta <- 2
F2 <- (1+Fscore_beta^2)*TruePositive /
    ((1+Fscore_beta^2)*TruePositive + Fscore_beta^2*FalseNegative + FalsePositive)

# Add data to a new row
csv[nrow(csv)+1,] <- list(Technique, TrueNegative, TruePositiveLow,
                          TruePositiveMed, TruePositiveHigh, FalsePositive,
                          FalseNegativeLow, FalseNegativeMed, FalseNegativeHigh,
                          Accuracy, Specificity,
                          Recall, RecallL, RecallU, Precision, PrecL, PrecU,
                          NPV, F1, F1L, F1U, Fhalf, F2)

# Save results
```

```
    tryCatch({write.csv(csv, paste(getwd(), location, "/resultsSummary.csv", sep=""),
    ↪  row.names=FALSE)}
             , warning=function(w) {print("The resultsSummary.csv file is open, unable to save.
             ↪  Run manual logResults() instead.")})
}


logResults <- function(df) {
    # Manually updated. Just a quick function to calculate success measure without
    # needing to run the entire model again, which takes a while.
    logResultsSummary(df,"Baseline - Always Defect","baselineDefectAll")
    logResultsSummary(df,"Baseline - Never Defect","baselineDefectNone")
    logResultsSummary(df,"Baseline - Random","baselineDefectRandom")
    logResultsSummary(df,"Second-Order Polynomial Regression - 80% CI","polyDefect_0.8")
    logResultsSummary(df,"Second-Order Polynomial Regression - 90% CI","polyDefect_0.9")
    logResultsSummary(df,"Second-Order Polynomial Regression - 95% CI","polyDefect_0.95")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99% CI","polyDefect_0.99")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.9% CI","polyDefect_0.999")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.92% CI","polyDefect_0.9992")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.94% CI","polyDefect_0.9994")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.96% CI","polyDefect_0.9996")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.98% CI","polyDefect_0.9998")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.999% CI","polyDefect_0.99999")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.9999%
    ↪  CI","polyDefect_0.999999")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.99999%
    ↪  CI","polyDefect_0.9999999")
    logResultsSummary(df,"Second-Order Polynomial Regression - 99.999999%
    ↪  CI","polyDefect_0.99999999")

    logResultsSummary(df,"Second-Order Polynomial Regression - 99% CI and 20X multiplier and
    ↪  window 6","polyDefect_0.99_multiplier20_window6","validation")
    logResultsSummary(df,"ARIMA(2,2,0) - 99% CI and 600x and window
    ↪  8","arima.2.2.0.Defect_600X.and.window.8","validation")

}
```

```
# Unfortunately, this step is performed manually for now...
library(boot)

# Yeah this is dumb, but uncomment one of these at a time and run it
column <- "polyDefect_0.8"
#column <- "polyDefect_0.9"
#column <- "polyDefect_0.95"
#column <- "polyDefect_0.99"
#column <- "polyDefect_0.999"
#column <- "polyDefect_0.9992"
#column <- "polyDefect_0.9994"
#column <- "polyDefect_0.9996"
#column <- "polyDefect_0.9998"
#column <- "polyDefect_0.99999"
#column <- "polyDefect_0.999999"
#column <- "polyDefect_0.9999999"
#column <- "polyDefect_0.99999999"
#column <- "polyDefect_0.99_multiplier20_window6"
#column <- "arima.2.2.0.Defect_600X.and.window.8"
#column <- "nnetDefect"
#column <- "metaAlg"

F1.fun <- function(df,idx)
{
    data <- df[idx,]

    TrueNegative <- nrow(data[which(data$rowDefect==0 & data[,column]==0), ])
    TruePositiveLow <- nrow(data[which(data$rowDefect==1 & data[,column]==1 &
    ↪  data$defectSeverity=="Low"), ])
    TruePositiveMed <- nrow(data[which(data$rowDefect==1 & data[,column]==1 &
    ↪  data$defectSeverity=="Med"), ])
    TruePositiveHigh <- nrow(data[which(data$rowDefect==1 & data[,column]==1 &
    ↪  data$defectSeverity=="High"), ])
    FalsePositive <- nrow(data[which(data$rowDefect==0 & data[,column]==1), ])
    FalseNegativeLow <- nrow(data[which(data$rowDefect==1 & data[,column]==0 &
    ↪  data$defectSeverity=="Low"), ])
    FalseNegativeMed <- nrow(data[which(data$rowDefect==1 & data[,column]==0 &
    ↪  data$defectSeverity=="Med"), ])
    FalseNegativeHigh <- nrow(data[which(data$rowDefect==1 & data[,column]==0 &
    ↪  data$defectSeverity=="High"), ])

    TruePositive  <- TruePositiveLow + TruePositiveMed + TruePositiveHigh
    FalseNegative <- FalseNegativeLow + FalseNegativeMed + FalseNegativeHigh

    Fscore_beta <- 1
    Fscore <- (1+Fscore_beta^2)*TruePositive /
        ((1+Fscore_beta^2)*TruePositive + Fscore_beta^2*FalseNegative + FalsePositive)
}

set.seed(0)

bootstrap <- boot(df, F1.fun, R = 1000)

boot.ci(boot.out = bootstrap, type = "norm")
```