Summer Internship Report

Sensor Anomaly Detection for Nuclear Reactor Systems Utilizing Linear Regression and K-Means Unsupervised Machine Learning

August 10, 2023

INL/RPT-23-74176 Rev:000

Liam M. Pohlmann[1,2]
Mentor: Dr. Palash Kumar Bhowmik[2]
Mentor/Manager: Dr. Piyush Sabharwall[2]

[1]University of New Mexico
[2]Idaho National Laboratory

Summer Internship Report

Sensor Anomaly Detection for Nuclear Reactor Systems Utilizing Linear Regression and K-Means Unsupervised Machine Learning

INL/RPT-23-74176 Rev:000

Liam M. Pohlmann[1,2]

Mentor: Dr. Palash Kumar Bhowmik[2]

Mentor/Manager: Dr. Piyush Sabharwall[2]

[1]University of New Mexico
[2]Idaho National Laboratory

August 10, 2023

Idaho National Laboratory
Irradiation Experiment Thermal Hydraulics Analysis
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

Page intentionally left blank

ABSTRACT

Nuclear reactors and related systems are becoming increasingly complex due to advancing technologies in next-generation power reactors. This increased complexity necessitates enhanced automation and data management capabilities. To successfully realize autonomous systems, methods must be developed to handle vast volumes of data and effectively distinguish anomalous data from noise and expected data.

While impressive models utilizing digital twins and similar approaches are under development, here we propose a simplified model for analyzing fundamental methods and techniques. Initially, we created a general dataset by using initial data from PCTRAN in order to represent ideal steady-state conditions. We then inserted anomalies based on prevalent sensor anomaly types (e.g., point anomalies, linear drift, and downward deviations), along with unusual anomalies such as exponential drift and upward deviations.

To detect anomalies, we developed a program that employs data partitioning and linear regression to preprocess and filter the anomalous data. A K-Means machine learning (ML) method was then applied to separate and count the data within the anomalous partition. The results from all datasets—apart from exponential growth—demonstrated positive outcomes, with each returning multiple instances of greater-than-95% accuracy.

We conducted further investigations using Idaho National Laboratory's RAVEN software to perform a sensitivity analysis on the input variables (R^2 Tolerance, Slope Tolerance, and Window Size) and found that the output variables (Accuracy and Time) were most sensitive to the Window Size.

Despite the promising results published, further development is required to effectively apply these methods to nuclear systems. Nevertheless, the strengths of this approach are evident and hold promise for future applications in the field.

CONTENTS

1 INTRODUCTION

With increasingly complex systems becoming ubiquitous within the nuclear energy sector, the importance of reliable sensor functionality and data greatly increases in regard to the successful operation of power plants [1]. Nuclear power generation in a fission reactor occurs through the release of energy produced by neutrons colliding with fuel atoms, making it often difficult (and occasionally *impossible*) to manually check or replace sensors within the reactor, even during a shutdown. Additionally, the complexity and number of sensors yield an overwhelmingly large quantity of data that cannot be analyzed manually. For this reason, we employed digital anomaly detection methods to report errors found within the sensor data.

Digital anomaly detection methods must be designed to be robust enough to detect subtle anomalies (e.g., drift), yet still perform quickly enough to alert workers the moment an issue is detected. However, if the system is *too* sensitive, it may report false positives, potentially leading to a plant shutdown that could cost thousands of dollars in lost revenue. On the other hand, if the system is not sensitive *enough*, plant issues may not be reported prior to the occurrence of a major event. Due to the dependencies within reactor systems, ensuring nuclear safety requires timely and accurate anomaly detection in the daily condition monitoring of Nuclear Power Plants (NPPs). Any slight anomaly in a plant could result in an irreversible and serious accident, along with high costs of maintenance and management [2].

In the beginning stages, anomaly detection is primarily concerned with data science. We are interested in developing a general process by which we can analyze many different sensors and achieve similar efficacy and efficiency. For the purposes of this report, we are strictly analyzing steady-state data, and this fact is heavily leveraged in the development of numerical and machine learning (ML) processes.

While a useful tool, ML tends to be computationally demanding and therefore inefficient in regard to both time and resources. Therefore, we looked to other methods in order to determine the necessity of ML methods. The problem can be phrased thusly: Our sensor-generated dataset is assumed to be steady-state, and thus approximately the same value, and yet we must develop a method for separating out anomalies in these data. A constant value across multiple time stamps would indicate a linear trend with a zero slope, suggesting that the model lends itself to linear regression (see Section 2.4), which offers additional statistical properties usable to determine whether the hypothesized linear fit is, in fact, a proper model for the data.

However, we will presume that extreme point anomalies are not the only anomalies found within the dataset. Therefore, we must account for this by not allowing the linear regression to examine the entire system, as this would not only lead to inaccuracies, but also increased computation time as the system grows. For these reasons, we sought to partition the collected data into *windows* of selected sizes small enough to guarantee a quick calculation time.

If, after these methods are employed, the system determines the existence of anomalies, we will then look to a simple *unsupervised*[1] ML method to identify those anomalies. Proper scaling methods were employed to guarantee success and reduce computation times (see Subsection 2.5 for details).

Similar methods, such as those seen in [3] and [4], utilize both clustering and linear regression to generate predictions of real-time anomalous data and predicted measurement values, respectively. The clustering methods in [3] similarly seek to partition the data, though they differ in their subsequent steps.

[1]Unsupervised methods are preferred due to their ability to be generalized for many different types of sensors and readings, without training.

2 METHODOLOGY

2.1 Overview

For this project, we sought to design a general process that enables *any* sensor data to be inputted and anomalies to be reported quickly and accurately. Figure 1 shows the steps followed in any data analysis process.

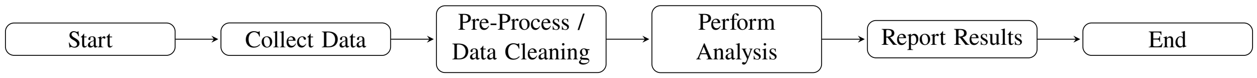


Figure 1: General Analysis Flowchart

The analyses covered in this report follow a similar, albeit more involved, process. Due to the challenging and varying nature of anomalies, *two* separate cleaning and analysis steps were employed. The first aims to check for the existence of anomalies, which can arise in the form of either point anomalies or drift anomalies. During this stage, the process involves an initial normalization of sensor values, the assignment of input values, and the removal of irrelevant data.

If anomalies are detected in the first analysis, the data are then subjected to further processing. This includes normalizing both the input *and* output values before utilizing ML (i.e., unsupervised K-Means analysis) to determine uniqueness and record point anomalies. Drift anomalies, however, cannot be separated using clustering methods; thus, the entire window will be recorded as anomalous.

Figures are included throughout this text to illustrate the transformation methods used to guarantee success for a generalized dataset. Changes in the data appear most drastic in later scaling (discussed in Section 2.5), as is required for the code to be successful.

2.2 Data Assignment

For the upcoming window analysis, we must define our input and output values. Each data point is assigned to a coordinate, with the dependent variable being the output and the independent variable being the index within the window[2]. Mathematically, we define:

$$X = \{x_1, x_2, x_3, \ldots, x_n\} = \{0, 1, 2, \ldots, n-1\} \tag{1}$$

and

$$Y = \{y_1, y_2, y_3, \ldots, y_n\}. \tag{2}$$

The graph generated by these assignments is illustrated in Figure 3. We then proceed with the analysis by applying localized linear regression.

[2]Example: The third data point, which reads $290K$, would be assigned the coordinate $(2, 290)$.

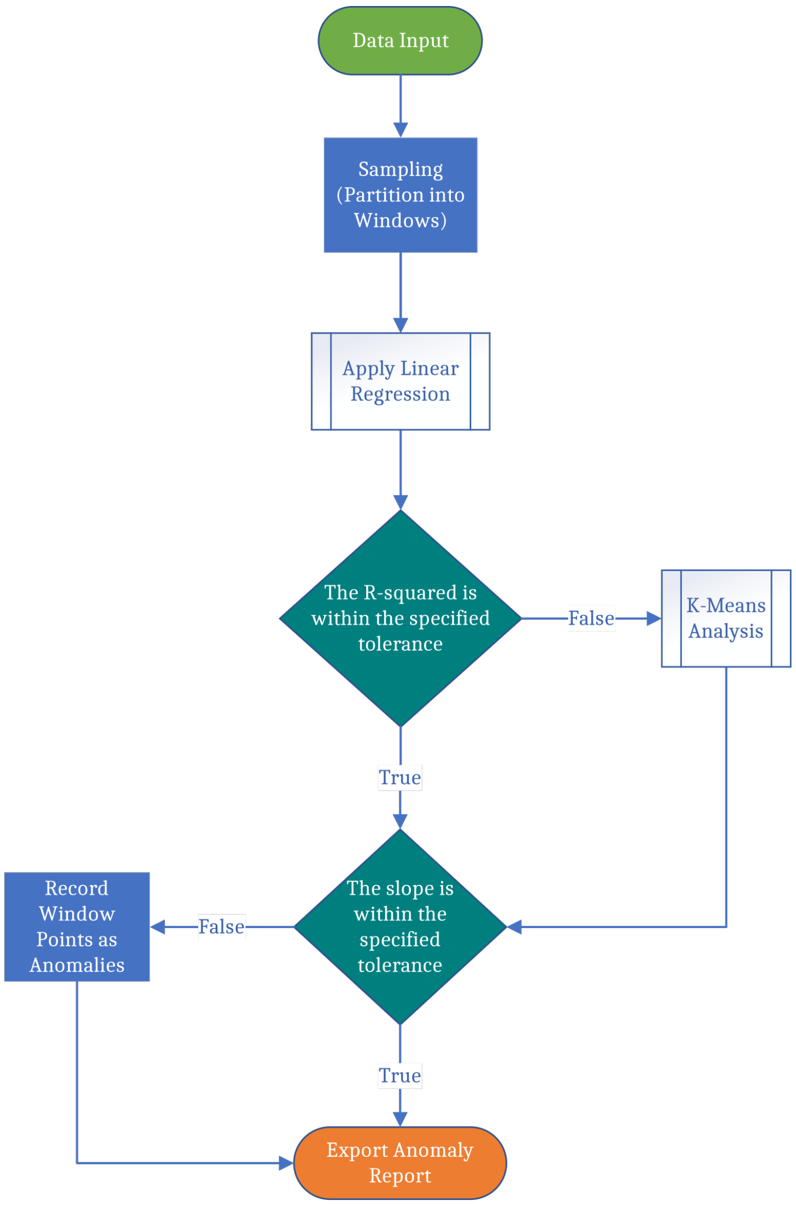

Data Input
Sampling (Partition into Windows)
Apply Linear Regression
The R-squared is within the specified tolerance
False
K-Means Analysis
True
The slope is within the specified tolerance
False
Record Window Points as Anomalies
True
Export Anomaly Report

Figure 2: Data Analysis Flowchart

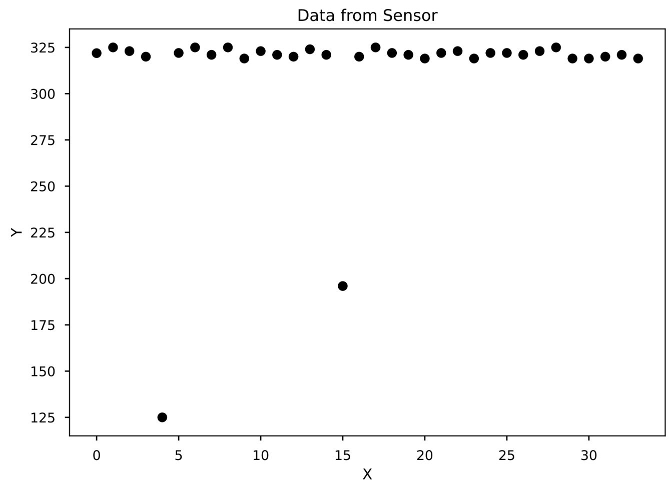


Figure 3: Sample Data from the Sensor

2.3 Data Sampling: Window Partitioning

One of the main obstacles in anomaly detection is the large quantity of data supplied—a quantity often too extensive to analyze manually *or* digitally. Processing such massive datasets can lead to decreased accuracy and increased computation times, resulting in belated and often overfitted results. To address this issue, we adopted a strategy of analyzing the data in smaller arrays (i.e., windows).

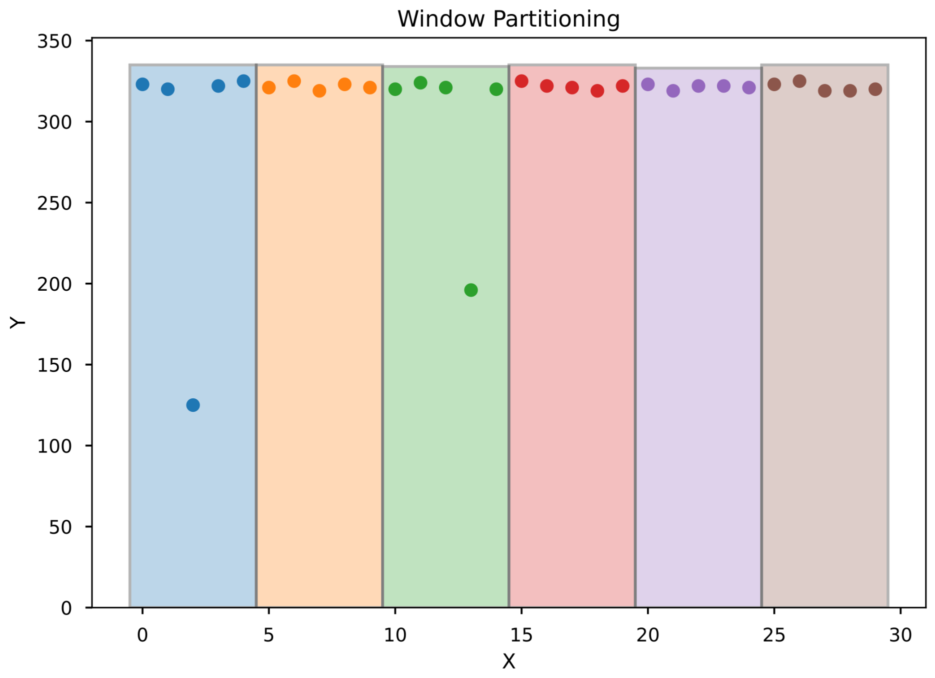


Figure 4: Window Partitioning

The size of the window (i.e., the number of data points within each window) is determined through experimental trials using datasets containing a known number of anomalies. By reducing the number of data points in each window, we achieve both increased computational speed and enhanced accuracy in the subsequent processes described in Subsections 2.4 and 2.5.

To establish an optimal window size, we created a range of window sizes and then iterated over them. The maximum window size was set to approximately one-fourth of the total number of sensor data points. Meanwhile, the minimum window size was fixed at 5 and incremented by 1 in each iteration. Hence, for any given iteration, the window size falls within the following range:

$$5 \leq \text{window size} \leq \left\lfloor \left(\frac{\text{number of data points}}{4} \right) \right\rfloor . \tag{3}$$

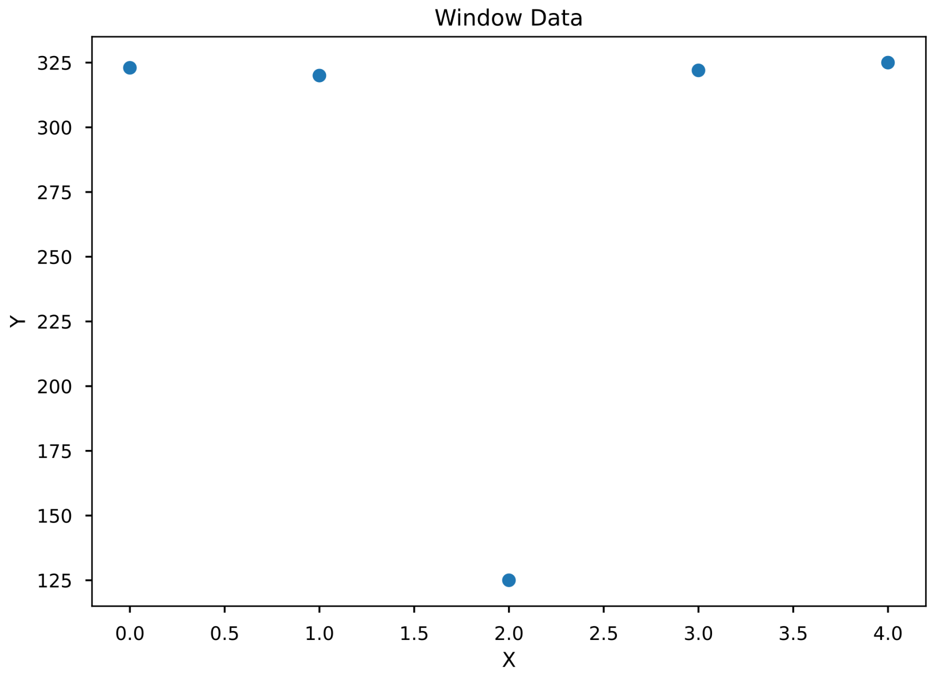


Figure 5: Window Data Only

2.4 Linear Regression

Linear regression is a process used to fit an overdetermined system (e.g., a scatter plot generated by sensor data) to a linear model ($y = mx + b$) (see Figure 8). The fitting is achieved by minimizing the residual vector, defined as:

$$\mathbf{r} = \mathbf{b} - A\bar{\mathbf{x}}, \tag{4}$$

where $A\bar{\mathbf{x}}$ represents the least-squares solution. Instead of manually establishing the *normal equations*, which involve properties of transpose and perpendicularity [5], we used tools available in the Python programming language to handle the linear algebra computations. Specifically, we used the `sklearn` library after installing it with the package manager `pip`, then importing to the script with: `from sklearn import metrics` [6].

Applying our anomaly detection method within a given window requires both the *coefficient of determination* and the *slope* of the regression; the equation of the least-squares solution and the plot are unnecessary for our analysis. Though the mathematics and tools for linear regression are well developed, many steps are required before such tools can be implemented. Figure (6) outlines the general logic and steps for the subprocess given in Figure 2.

2.4.1 Slope Analysis

The slope analysis of the generated linear regression is foundational to this report, as it and the R^2 value determine the existence of anomalies within a data window. The primary concern in this report (see Subsection 2.4.3 for details) lies in how close the slope in any window is to 0, as large deviations will indicate the

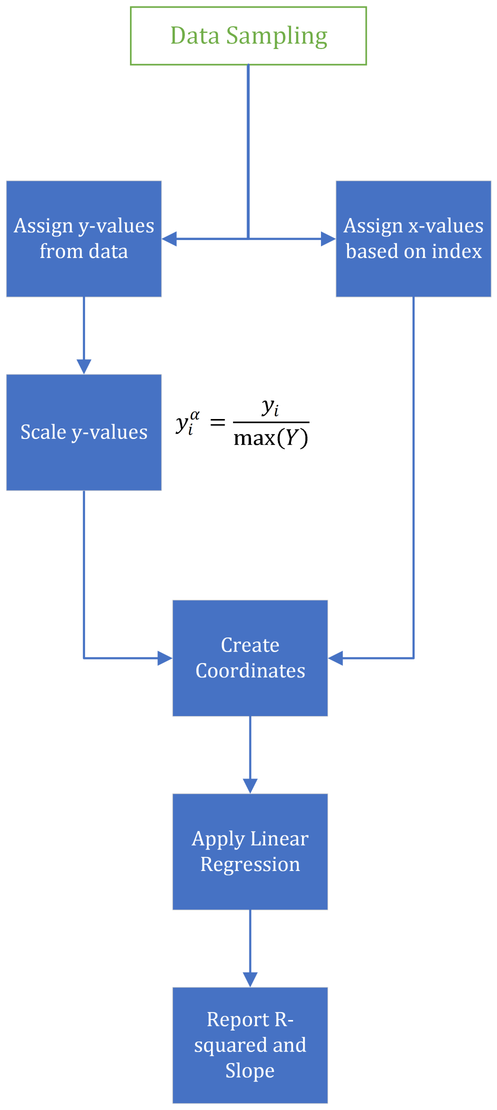

Data Sampling
Assign y-values from data
Assign x-values based on index
Scale y-values
$y_i^\alpha = \frac{y_i}{\max(Y)}$
Create Coordinates
Apply Linear Regression
Report R-squared and Slope

Figure 6: Linear Regression Subprocess Flowchart

presence of anomalies. The anomaly type is determined by a combination of variables, including the coefficient of determination. Although this report emphasizes anomaly detection, the classification of anomalies is not its main focus.

To ensure that the slope bounds are universally applicable for all sensors and measurements, we normalized the measured data. This normalization process scales the values without altering the relationship between points. We achieved this by dividing each window value by the maximum value within the window, defined as follows:

$$y_i^\alpha = \frac{y_i}{\max(Y)}. \tag{5}$$

This results in the normalized window:

$$Y_\alpha = \{y_1^\alpha, y_2^\alpha, \ldots, y_n^\alpha\}. \tag{6}$$

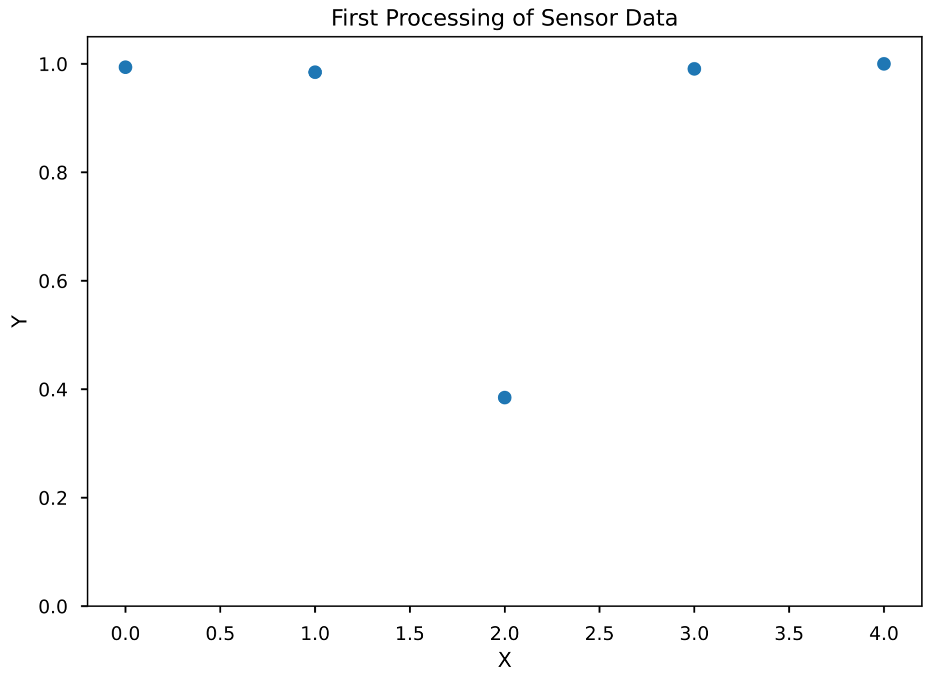


Figure 7: Scaled: Y-values

Next, we applied linear regression to the normalized data, using the given input values (in this case, the index values). For a visual representation of linear regression when one anomaly is present, see Figure 8.

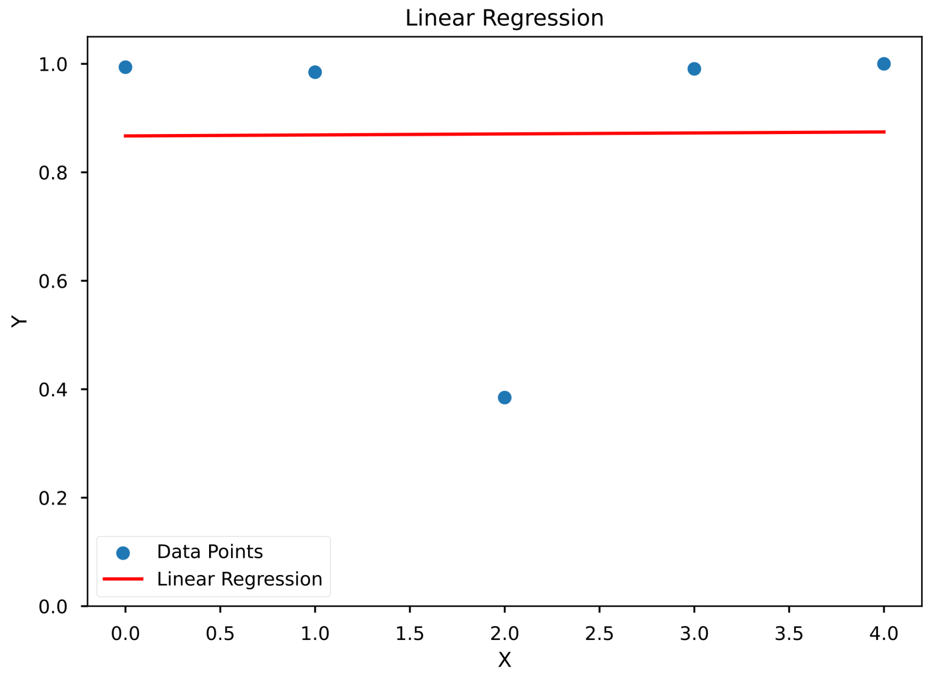


Figure 8: Linear Regression of a Window

2.4.2 Coefficient of Determination

As we were interested in assessing the fit of the regression model onto the data within the window, we needed a method of determining the “goodness” of the fit. For this purpose, we employed the coefficient of determination, also known as R^2, which is defined as follows:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}, \tag{7}$$

where $\hat{y}_i$ represents the value predicted from the regression and $\bar{y}$ is the mean of the window [7].

Closer inspection revealed that, as the regression model better fits the data (meaning $\hat{y}_i \rightarrow y_i$), the R^2 value approaches 1. Hence, we generally searched for a regression model (e.g., linear, quadratic, or trigonometric) that maximized this value. For the cases in this report, we exclusively used *linear* regression.

Additionally, it is worth noting that the denominator of this expression poses a potential concern. If the mean of the dataset is very close to the data points (meaning the data points carry approximately the same value for all inputs), the denominator sum approaches 0, resulting in an undefined R^2 value. While undefined values usually result from program errors, we leveraged this fact when filtering the data (see Subsection 2.4.3 for further details).

2.4.3 Model Fitting: Application

The analysis of steady-state sensor data required different standards for evaluating R^2 and slope values. As discussed in Subsection 2.4.2, the coefficient of determination can become unstable—and eventually undefined—when the mean of the sample approaches the value of the data points. This occurs when all values in the window are identical:

$$y_1 = y_2 = \ldots = y_n. \tag{8}$$

Or put more simply:

$$\min(Y) = \max(Y), \tag{9}$$

indicating a perfect steady-state case.

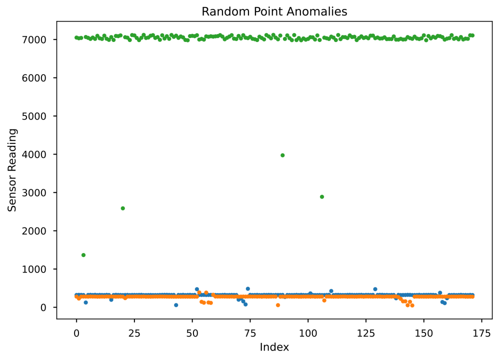
To effectively identify anomalies in steady-state data, we relied primarily on the slope of the regression. The data points were *approximately* equal to each other when both R^2 and the slope were approximately 0, thus we modified our program to specifically search for *small* R^2 and slope values.

Based on this logic, we defined parameters for both point and drift anomalies, with the former being easier to separate and count. Point anomalies are recorded when using K-Means clustering (see Subsection 2.5) if the magnitude of the slope exceeds the given bounds *and* the R^2 value is sufficiently small. Linear drift anomalies are recorded if both the magnitude of the slope and R^2 exceed the specified bounds, implying a linear change in values over time. Additionally, we explored exponential drift anomalies, as identified by a large slope and a small R^2 value. Table 1 and Figures 9 and 10 list the various anomalies of interest.

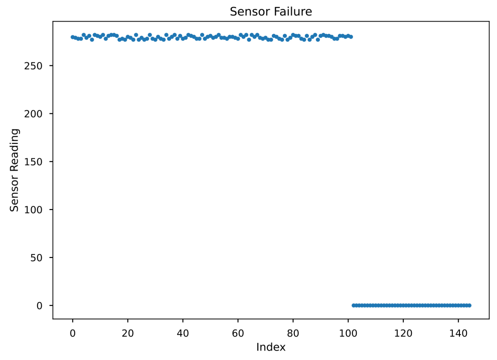
Type	**Common Anomalies**	**Unusual Anomalies**[3]
1	Point Anomalies	10% *Up* Deviation
2	Sensor Break	5% *Up* Deviation
3	Erratic Failure	Exponential Drift
4	10% *Down* Deviation	
5	5% *Down* Deviation	
6	Linear Drift	

Table 1: Anomalies Examined for the AP1000-like System

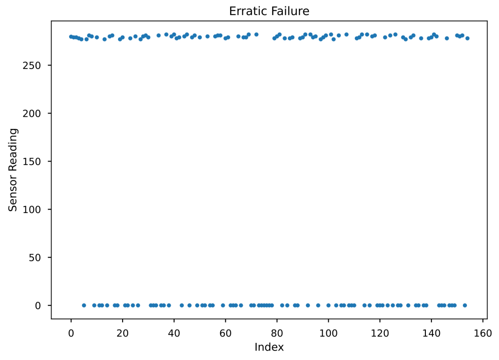
[3]See Appendix for results.



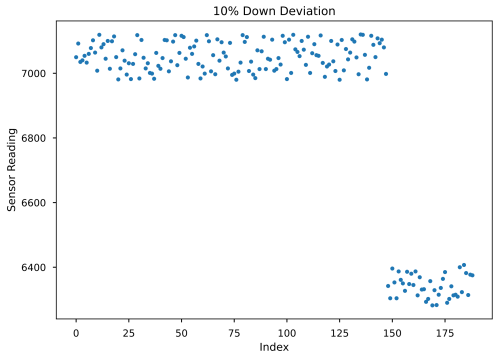
(a) Common Anomaly Type-1



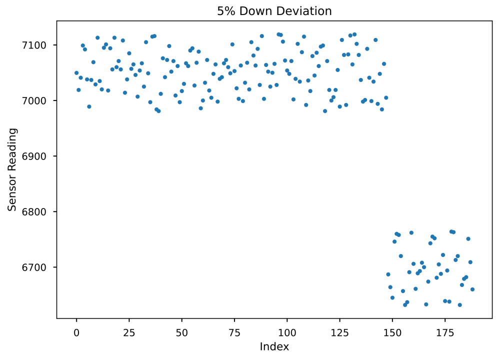
(b) Common Anomaly Type-2



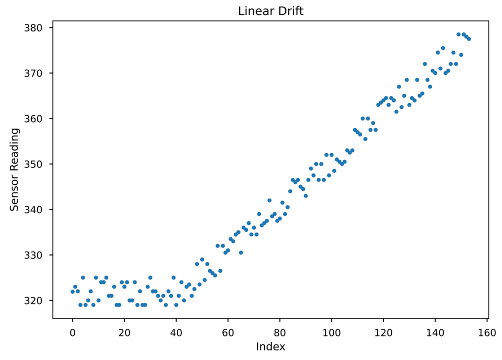
(c) Common Anomaly Type-3



(d) Common Anomaly Type-4

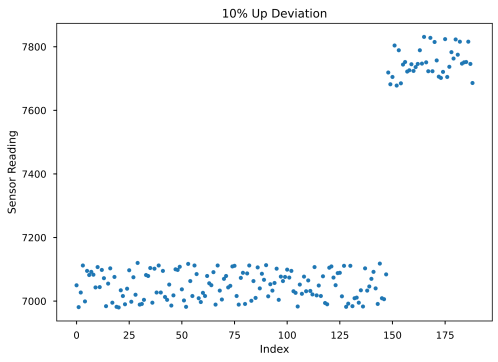


(e) Common Anomaly Type-5

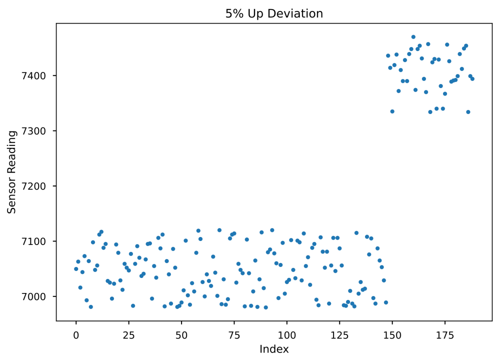


(f) Common Anomaly Type-6

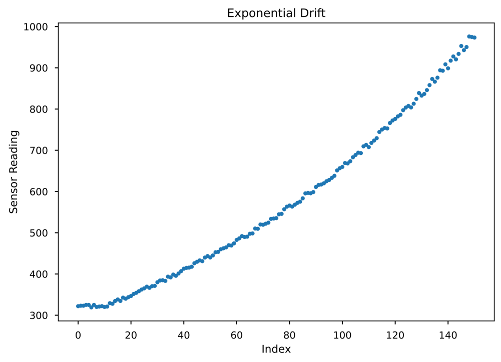
Figure 9: Plots of Common Anomalies



(a) Unusual Anomaly Type-1



(b) Unusual Anomaly Type-2



(c) Unusual Anomaly Type-3

Figure 10: Plots of Unusual Anomalies

2.5 K-Means Clustering

K-Means clustering is an unsupervised ML algorithm that groups together "clusters" data, based on their distances from generated points (i.e., "centroids"). These points are randomly generated for the first iteration, then redefined based on the sum of squared errors (SSE). The algorithm runs until either reaching a predetermined number of iterations or until the SSE has converged to its minimum. The process results in data that are labeled based on the nearest centroid, effectively grouping together similar data [8].

However, K-Means is not completely hands-off, as the algorithm requires an input for the number of clusters (k). To optimize the k-value, we would typically use methods such as the elbow method[4]. However, in this case, we scaled the data such that the program only required two ($k = 2$) clusters to guarantee success.

Once the data had failed the original coefficient of determination (R^2) test, we worked to implement K-Means clustering in order to group together anomaly data, then store and report the number of anomalies found. Though simply stated, ensuring the efficacy of the clustering required additional steps in order to prepare the data prior to the K-Means analysis. This report explores in depth the data normalization methods applied to all the windows and datasets. For a visual representation of the K-Means subprocess, see Figure 11.

[4]See Appendix.

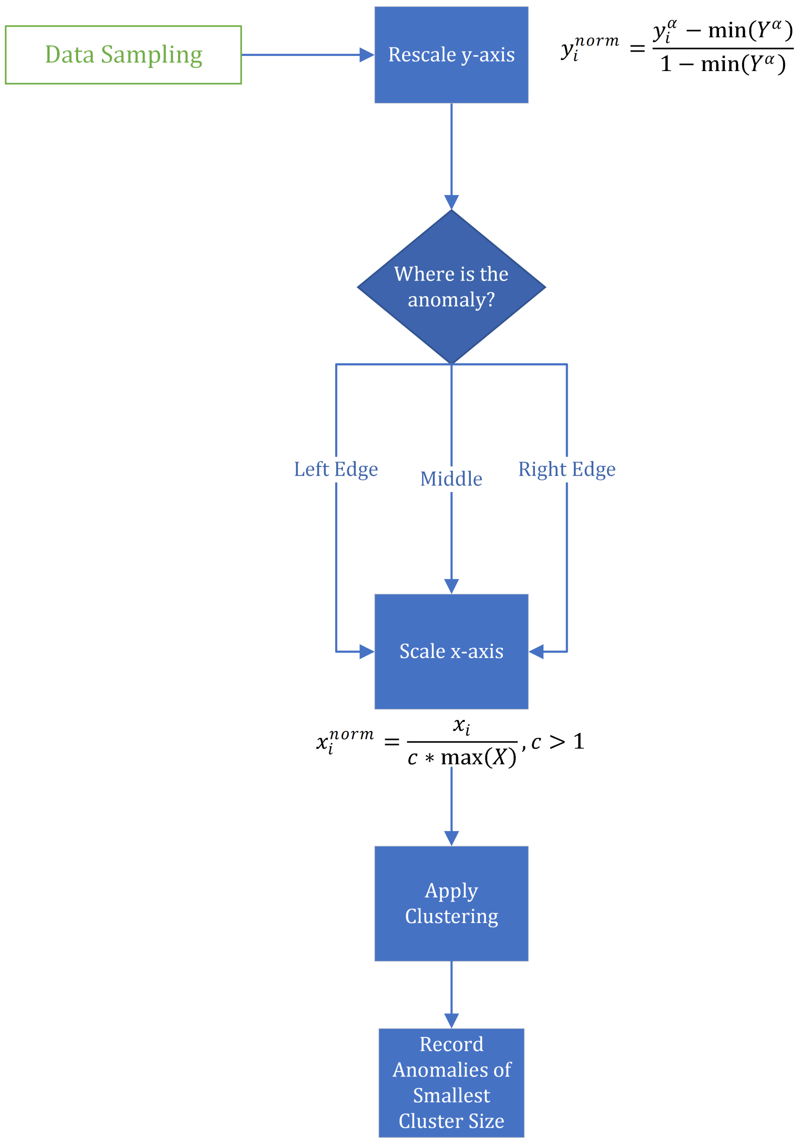

Data Sampling
Rescale y-axis
$y_i^{norm} = \frac{y_i^\alpha - \min(Y^\alpha)}{1 - \min(Y^\alpha)}$
Where is the anomaly?
Left Edge
Middle
Right Edge
Scale x-axis
$x_i^{norm} = \frac{x_i}{c * \max(X)}, c > 1$
Apply Clustering
Record Anomalies of Smallest Cluster Size

Figure 11: K-Means Subprocess Flowchart

2.5.1 Y-axis Scaling

In preparing the data, we first looked at the dependent variable. Since we were interested in making the anomaly data as evident as possible, we sought a method of scaling the axis while also separating out the anomaly data. To meet these requirements, we derived, for $1 \leq i \leq n$, the following simple method:

$$y_i^{norm} = \frac{y_i^\alpha - \min(Y_\alpha)}{\max(Y_\alpha) - \min(Y_\alpha)} \tag{10}$$

Or more simply:

$$y_i^{norm} = \frac{y_i^\alpha - \min(Y_\alpha)}{1 - \min(Y_\alpha)}, \tag{11}$$

since $\max(Y_\alpha) = 1$.

This normalized set, Y^{norm}, has effectively separated the anomaly values by exacerbating the differences, regardless of their original scale (see Figure 12).

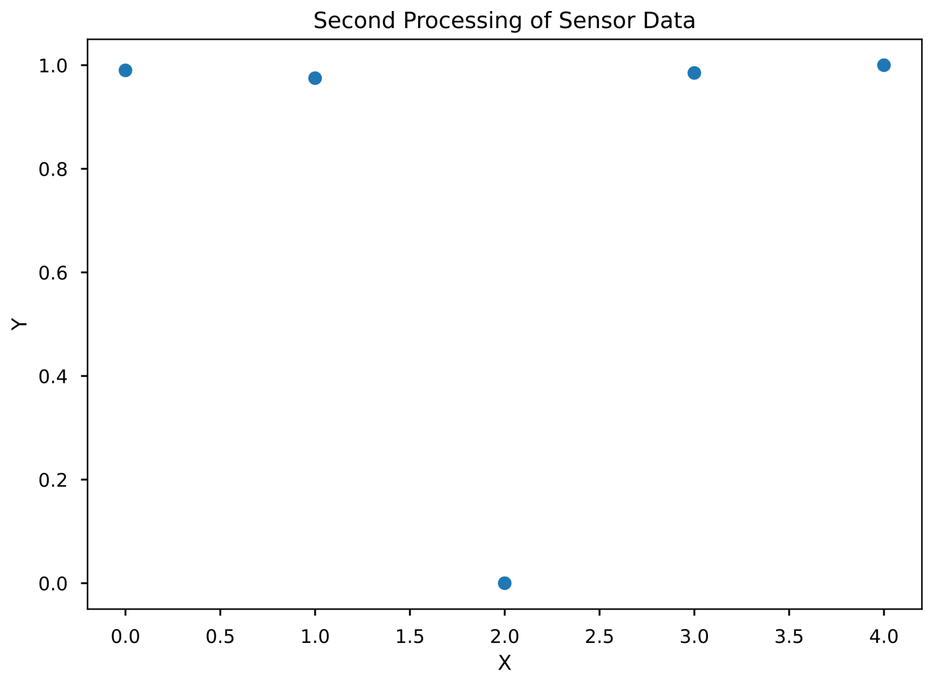


Figure 12: Normalized Y-axis

2.5.2 X-axis Scaling

The independent variable, however, proved more complicated. Though we had normalized the dependent variable, we were still left in the same situation as before, in that we had not yet guaranteed the success of the K-Means clustering. To provide such a guarantee, we needed to create conditions such that the shortest distance from an anomaly always exceeded the farthest distance from any two non-anomaly data points, or:

$$\begin{aligned} &\min\left(d(P_p, P_k)\right) > \max\left(d(P_k, P_j)\right); \\ &\qquad k \in \{0, 1, 2, \ldots, n; k \neq p\}, \\ &\qquad j \in \{0, 1, 2, \ldots, n; j \neq p; j \neq k\}, \end{aligned} \tag{12}$$

where the distance between two points, $P_a = (x_a, y_a)$ and $P_b = (x_b, y_b)$, is calculated as:

$$d(P_a, P_b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2} = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}. \tag{13}$$

For this calculation, we employed the index (X) values to change the distance relationships between anomaly points and the neighboring non-anomaly data. We considered three unique cases (see Table 2).

In all cases, we normalized the x-values by dividing by the largest, such that:

$$x_i^{norm} = \frac{x_i}{\max(X)}. \tag{14}$$

Case Number	Symbolic	Interpretation
1	$P_p = P_1$	Anomaly is on the left-most edge
2	$P_1 \neq P_p \neq P_n$	Anomaly is located between the edges
3	$P_p = P_n$	Anomaly is on the right-most edge

Table 2: Independent Variable Scaling Cases

This normalization enabled us to employ basic calculus to state the following:

$$\lim_{n\to\infty} d(P_p, P_{p\pm1}) = |y_p - y_{p\pm1}|. \tag{15}$$

This simple statement, via simple application of the distance formula (13), allowed us to bound our values as follows:

$$1 \leq \min(d(P_p, P_{p\pm1})) \leq \sqrt{2}. \tag{16}$$

Conservatively, we assumed the lower bound for all cases (i.e., $\min(d(P_p, P_{p\pm1})) = 1$).

Case I: $\mathbf{P_p = P_1}$

In this case, we know where each point is, so we can rewrite the inequality so as to be more clear:

$$\min(d(P_1, P_2) > \max(d(P_2, P_n)). \tag{17}$$

Using the distance formula gives us:

$$\max(d(P_2, P_n)) = \sqrt{\left(1 - \frac{1}{x_n}\right)^2 + (y_n - y_2)^2}. \tag{18}$$

This is because we assume only one anomaly: $(y_n - y_2)^2 \approx 0$. However, for a sufficiently large x_n, we see that Equation (18) will approach 1 quickly. We can bypass this by instead normalizing by $c * \max(X) = c * x_n$, where c is a constant greater than 1, yielding:

$$\max(d(P_2, P_n)) = \frac{1}{c}\left(1 - \frac{1}{x_n}\right), \tag{19}$$

which satisfies the inequality.

Case II: $\mathbf{P_1 \neq P_p \neq P_n}$

Using the same process as in **Case I**, we rewrite the desired inequality as follows:

$$\min(d(P_p, P_{p\pm1})) > \max(d(P_1, P_n)). \tag{20}$$

Given that $P_1 = 0$ and $P_n = 1$ (after normalization), we can use the distance formula to expand the statement to the following:

$$\sqrt{\left(\frac{x_p - x_{p\pm1}}{x_n}\right)^2 + \left(y_p - y_{p\pm1}\right)^2} > \sqrt{1 + (y_n - y_1)^2}. \tag{21}$$

We now use the fact that, for a large n, $\left(\frac{x_p - x_{p\pm1}}{x_n}\right) \approx 0, \min(y_p - y_{p\pm1}) = 1$, as well as $|y_n - y_1| \approx 0$ (under the assumption that only one anomaly exists in the window), reduce even further to $y_p - y_{p\pm1} \approx 1 > 1$, which is a false statement. To remedy this, we instead normalize by dividing by $c * \max(X) = c * x_n$, thus yielding the following:

$$|y_p - y_{p\pm1}| \approx 1 > \frac{1}{c}, \tag{22}$$

which satisfies the inequality.

Case III: $\mathbf{P_p = P_n}$

Using the same process and assumptions as implemented in **Case I** and **Case II**, we create the following inequality:

$$\min(d(P_n, P_{n-1})) > \max(d(P_1, P_{n-1})). \tag{23}$$

We then again look at the case in which $\min(d(P_p, P_{p\pm1})) = 1$. Substitution leads to the following:

$$\begin{aligned}\max(d(P_1, P_{n-1})) = \sqrt{\left(\frac{x_{n-1}}{x_n}\right)^2 + (y_{n-1} - y_1)^2} \\ = \frac{x_{n-1}}{x_n}.\end{aligned} \tag{24}$$

When substituting for $x_i = i - 1$ and normalizing instead by $c * \max(X) = c * x_n$, we find:

$$\max(d(P_1, P_{n-1})) = \frac{1}{c}\left(\frac{n-2}{n-1}\right), \tag{25}$$

which converges to $\frac{1}{c}$ for large n, again satisfying the inequality.

Conclusion

In this section, we aimed to ensure successful K-Means clustering by normalizing the x-axis values while maintaining separation between the anomalous and the expected data points. The conditions required for successful K-Means clustering were found to be satisfied by normalizing the x-values by using the maximum of the x-values in tandem with a constant, c, which is greater than 1, then further normalizing the y-values by using the maximum and minimum of the y-values. Considering the absence of the upper bounds necessary on c to guarantee efficacy in a ($k = 2$) K-Means clustering, we chose $c = 2$ for normalization within the code in order to foster simplicity and ease of implementation.

Figure 13 visually demonstrates the effect of this normalization process, highlighting the clear separation of anomaly data while preserving the overall data distribution.

By employing the normalization techniques described, we successfully prepared the data for K-Means clustering and improved the accuracy and reliability of our anomaly detection process.

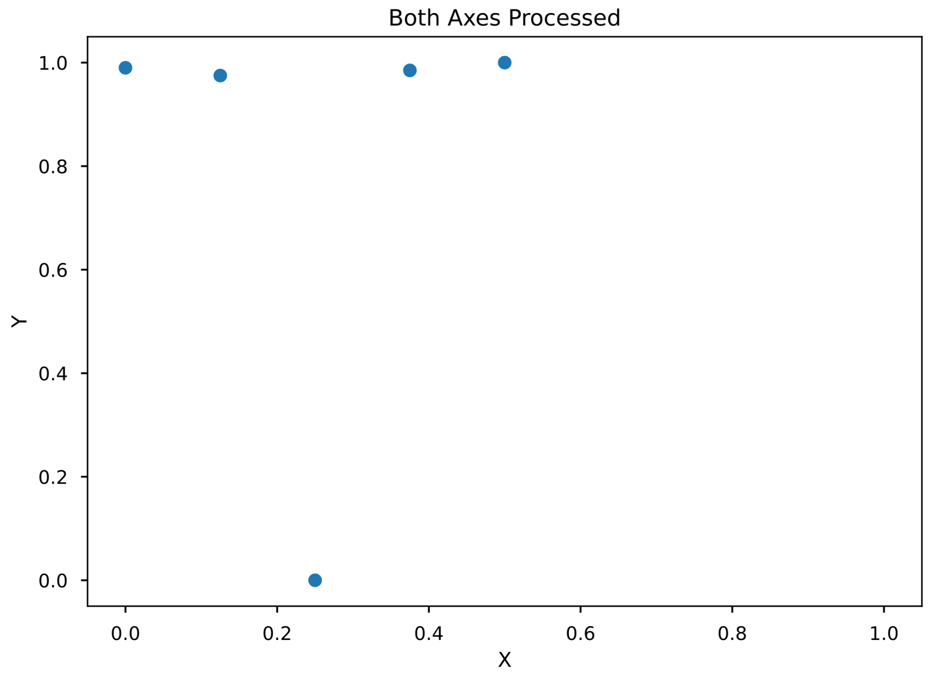


Figure 13: Normalized: X-axis and Y-axis

With the data now suitably preprocessed, we are ready to proceed with the K-Means clustering analysis in order to identify and record anomalies effectively. The next section delves into the results and implications of our anomaly detection process.

2.5.3 Clustering

With normalized axes to assure us of successful clustering, we implemented a $k = 2$ K-Means clustering that utilized the packages available in *Scikit-Learn*. Without previous training, the model can only separate the anomaly data from the normal data, without the ability to effectively dictate *which* are the anomaly data. Since we were assuming steady-state data, we did not implement artificial intelligence. Instead, we designed the program so as to record the smaller clusters as being anomalies, effectively providing a lower bound for the number of anomalies. Ideally, this bound enabled removal of the possibility of over-counting as a result of the K-Means method; rather, over-counting will be a result of false drift readings or false triggering of the K-Means process. See Figure 14 for a visual representation of the clustering technique.

3 REACTOR SYSTEM DATA

3.1 Simulation

To test the code's efficacy on different types of anomalies using simulator data. The simulator data were taken from PCTRAN [9], then perturbed by 1%. Specific anomalies were then added to different columns, assuming only one anomaly per row (for easier autonomous counting of inserted anomalies). The code was

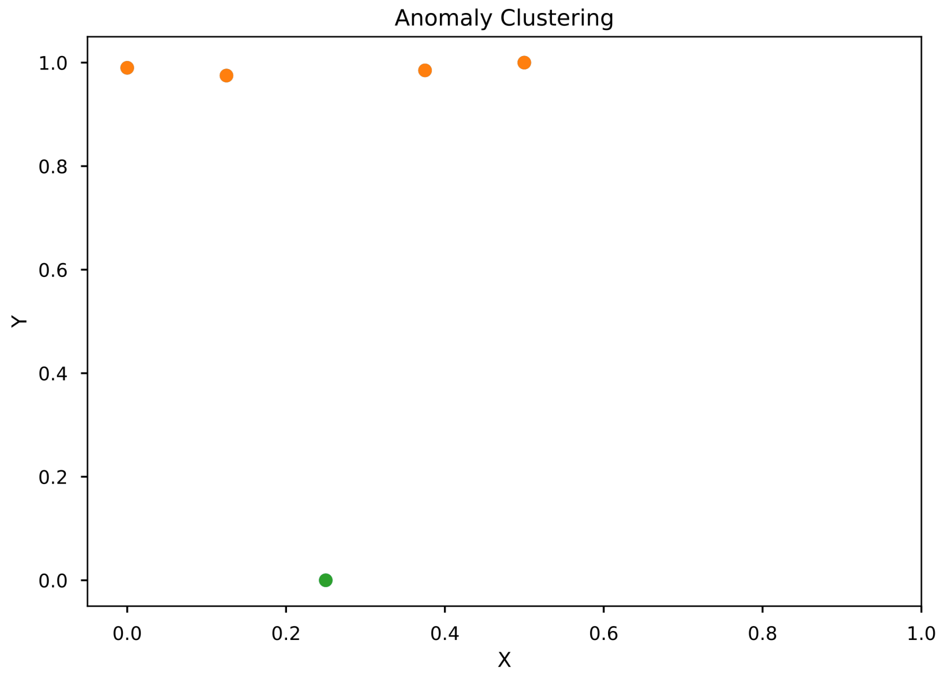


Figure 14: K-Means Clustering after Normalization

then run over a range of R^2 tolerances, slope tolerances, and window sizes, and a `.csv` file was exported listing the configurations that yield an accuracy of greater than 95%.

The purpose of the specialized datasets was to test the efficacy of the process by allowing the *type* of anomaly, not the quantity of anomalies, to act as the independent variable. By comparing the accuracy and parameters of all valid configurations, conclusions can be drawn as to the optimal configurations that will yield the greatest anomaly detection accuracy.

3.2 PCTRAN Steady-State AP1000 Reactor System

Relatively steady-state data were exported utilizing PCTRAN's [9] demo version, and a Python script was developed to randomly insert point anomalies into the system prior to the analysis. Unlike the datasets described in the previous section, the purpose of this dataset was *not* to test the efficacy of the program on anomalous data, but to provide a sensitivity analysis of the program by utilizing RAVEN (see Section 4.2 for more details).

4 RESULTS AND DISCUSSION

4.1 Simulator Data Analysis

As discussed, we analyzed specific anomaly cases in the RandAP1000 datasets (see Table 1 for all cases considered), which were artificially created using PCTRAN [9] and Microsoft Excel. The sample reports provided in Appendices B and C are actual results reported by the program for those configurations that yielded an accuracy of greater than or equal to 95%; however, many results were trimmed and condensed to fit this report. Also note that, while the program did very well at analyzing the common anomalies described in this report, it failed to report a viable accuracy in analyses for the unusual anomaly Exponential Drift.

Overall, the program performed very well in detecting all common anomalies and most (i.e., two out of three) cases of unusual anomalies, with most reporting many viable configurations and some even reporting 100% accuracy. These simulation data, however, are insufficient to prove efficacy in a true system, as the processes created only utilize mathematical properties of the data, not physical properties. Furthermore, the processes were specifically developed to handle cases involving strictly steady-state conditions, and each variable was only analyzed individually. Therefore, modifications to the process must be made in order to allow for the more dynamic environment of a nuclear reactor.

4.2 RAVEN-assisted Sensitivity Analysis

RAVEN is a risk analysis software developed by Idaho National Laboratory, designed for uncertainty quantification testing, generating reduced-order models, and conducting sensitivity analyses, among other applications [10]. Incorporating this software, we perturbed the input values of R^2 Tolerance, Slope Tolerance, and Window Size within predefined limits. These perturbed values were then utilized in a modified program that follows the same processes as described earlier. We focused our analysis on Accuracy and Time as these parameters hold particular significance in the context of nuclear systems. The procedure involved running the script 500 times with Monte Carlo sampling for R^2 Tolerance and Slope Tolerance. The allowable ranges were set as 0.01–0.1 and 0.05–0.5, respectively. Furthermore, for the variable Window Size, we employed discrete uniform sampling across the inclusive range of 5 to 20. Figure 3 displays the results of this sensitivity analysis, highlighting Window Size as the most influential factor for both Accuracy and Time. (Refer to Appendix E for the scripts employed in the calculations.) This observation is justifiable due to the direct impact of the window size on computations within the Linear Regression component of the methods, whereas the slope and R^2 tolerances function as straightforward boolean parameters.

Input	Output	Normalized Sensitivity
R^2 Tolerance	Accuracy	0.100
Slope Tolerance	Accuracy	-0.008
Window Size	Accuracy	0.208
R^2 Tolerance	Time	0.411
Slope Tolerance	Time	-0.131
Window Size	Time	-1.681

Table 3: Normalized Sensitivity Analysis Results

5 CONCLUSION AND PATH FORWARD

The conducted tests aimed to demonstrate the feasibility of the described methods, particularly the adoption of linear regression as a primary technique for assessing the need for machine learning (ML). The implemented code was tailored to assess processes applicable to transient datasets in real-life scenarios, although substantial modifications are essential to ensure proper coordination between sensors, the program, and personnel.

The demonstrated techniques have effectively identified common anomalies as explored in this study. However, achieving a genuinely resilient system capable of detecting rare anomalies and long-term drift necessitates further testing and refinement.

Nevertheless, the methodologies presented in this report do not effectively mitigate reactor system noise originating from routine operations. Consequently, there is a likelihood of *over*-detection, which, if unaddressed, could lead to substantial financial losses in cases where false anomaly alerts result in plant shutdowns. Furthermore, the methods devised rely on *data* rather than physics-driven approaches. While this facilitates quicker implementation at the expense of versatility, the ML model lacks training on physics models.

As the model assesses one sensor at a time, its efficacy hinges on reactor consistency to flag anomalies. Despite this limitation, the developed methods hold potential for various applications within reactor systems. Their robustness offers a balance between acceptable accuracy and impressive speed. Future development can lead to their enhancement and broader applicability.

6 ACKNOWLEDGEMENTS

This presentation is a culmination of work conducted over the course of a 10-week internship. The content and results would not have been possible without the support of the following individuals (listed in no particular order): Dr. Palash Bhowmik, Dr. Piyush Sabharwall, Dr. Congjian Wang, and Dr. Richard Skifton. Thank you for your counseling throughout my experience here at INL.

A ELBOW METHOD

Though not pertinent to this report, the elbow method is a technique used to determine the optimal k-value in a K-Means clustering ML algorithm.

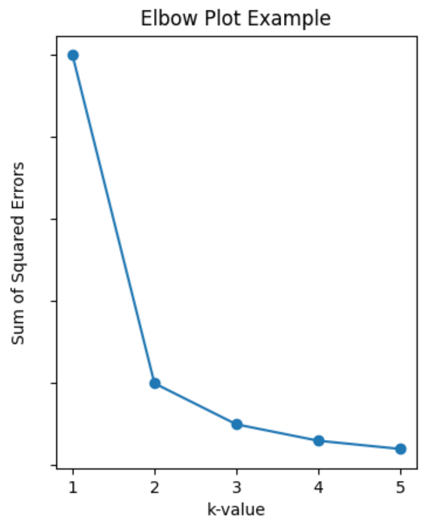


Figure 15: SSE vs. k

Plotting the sum of squared errors against k (see Figure 15) reveals a sharp change in the slope of the graph at $k = 2$—a sudden change that appears similar to the flex point of an elbow joint. In this method, we look for the sharpest "elbow" in the plot in order to determine the optimal number for k. Because this technique is somewhat subjective, we could simply *look* at the graph to determine what we believe k should be [11]; however, this would be unreasonable if the algorithm must be run multiple times. It is for this reason we propose the following method of consistently selecting the optimal k value.

If we were to take a moment to define what an elbow is with respect to this application, we would perhaps come up with such definitions as "the point at which the plot becomes relatively linear" or, less scientifically, "the deepest point." This paper argues that the elbow be defined as the point at which the distance from a point to a line projected between its two surrounding vertices is maximized (see Figure 16).

It is known that the shortest distance from a point to a line is the length of a line segment that is perpendicular to the line and terminates at the point. The distance between two points, $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, on a 2-D space is calculated using Pythagorean's theorem.

Looking at Figure 15, we let the point at $k = j$ be $P_3 = (x_3, y_3)$, the point at $k = j - 1$ be $P_1 = (x_1, y_1)$, and the point at $k = j + 1$ be $P_2 = (x_2, y_2)$. Lastly, we defined the point on the line closest to the P_3 as $P_0 = (x_0, y_0)$, and we were interested in computing the distance from P_3 to P_0:

$$d(P_0, P_3) = \sqrt{(x_0 - x_3)^2 + (y_0 - y_3)^2}. \tag{26}$$

Using basic linear algebra, we showed that the shortest distance from a point to the projected line can be represented symbolically, using the coordinates of each point, as:

$$d(P_0, P_3) = \sqrt{(x_0 - x_3)^2 + \left(\frac{y_1 - y_2}{x_1 - x_2}(x_0 - x_1) + y_1 - y_3\right)^2}, \tag{27}$$

for:

$$x_0 = \frac{\frac{y_1 - y_2}{x_1 - x_2} + \frac{x_1 - x_2}{y_1 - y_2} x_3 + y_3 - y_1}{\frac{y_1 - y_2}{x_1 - x_2} + \frac{x_1 - x_2}{y_1 - y_2}}. \tag{28}$$

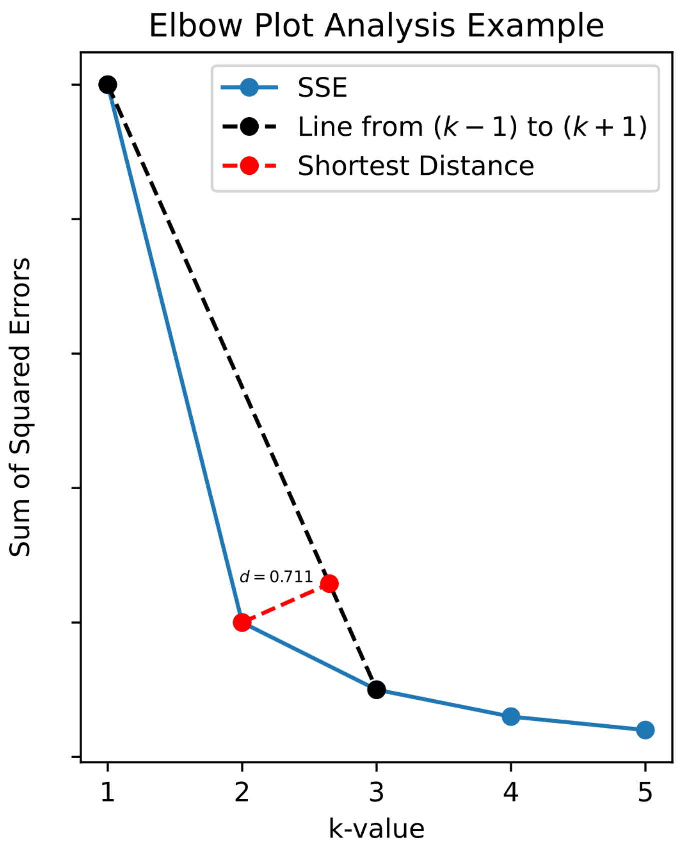


Figure 16: Elbow Method

In practice, we allow the program to run the K-Means algorithm for a range of k that would seem reasonable for the dataset (if there is any intuition available), then allow it to select the optimal k-value (using this program) and report the results.

A.1 Code

```python
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.cluster import KMeans
import os

# The purpose of this code is to provide an example framework for implementing
# the Elbow Method, as described in the report provided.

def ElbowMethod(DataFrame, KMax, FileName):
    FileNameEdited = os.path.splitext(FileName)[0]
    SumSqError = []
    distances = []
    k_range = range(1, KMax + 1)
    for k in k_range:
        km = KMeans(n_clusters=k)
        km.fit_predict(DataFrame[['HLT_A', 'CLT_A', 'Flow_A']])

        SumSqError.append(km.inertia_)

        if k - 2 > 0:
            tuples = [(k - 3, SumSqError[k - 3]), (k - 2, SumSqError[k - 2]), (k - 1,
                                                                                 SumSqError[k - 1])]
            new_distance = LinePointDistance(tuples[0], tuples[2], tuples[1])
            distances.append(new_distance)

    k_optimal = distances.index(max(distances)) + 2

    plt.xlabel('K')
    plt.ylabel('Sum of Squared Error')
    plt.title('Elbow Plot')
    plt.plot(k_range, SumSqError)
    plt.plot(k_optimal, SumSqError[k_optimal - 1], color='black', marker='o')
    plt.legend(['Sum of Squared Error', 'k-optimal'])

    K_str = str(KMax)
    plt.savefig(f'{FileNameEdited}/ElbowPlot{K_str}_{FileNameEdited}.svg')

    return k_optimal


def LinePointDistance(P1, P2, P3):
    # Function computes the distance between the line defined by points P1, P2 and the point P3
    x1 = P1[0]
    x2 = P2[0]
    x3 = P3[0]

    y1 = P1[1]
    y2 = P2[1]
    y3 = P3[1]

    m_12 = (y1 - y2) / (x1 - x2)
    m_12_inverted = m_12 ** (-1)

    x0 = (m_12 * x1 + m_12_inverted * x3 + y3 - y1) / (m_12 + m_12_inverted)
    left_side = x0 - x3
    right_side = m_12 * (x0 - x1) + y1 - y3

    distance = (left_side ** 2 + right_side ** 2) ** (1 / 2)
    return distance
```

B COMMON ANOMALY TEST CASES

SAMPLE Random Point Anomalies				
R^2 Tolerance	**Slope**	**Window Size**	**Accuracy**	**Time**
0.01	0.1	5	95.12	0.22
0.01	0.1	8	95.12	0.22
0.01	0.1	12	97.56	0.25
0.01	0.15	12	97.56	0.24
0.02	0.05	22	100	0.14
0.02	0.05	30	97.56	0.16
0.02	0.05	32	95.12	0.12
0.02	0.1	11	100	0.25
0.02	0.1	22	100	0.11
0.03	0.25	22	97.56	0.19
0.03	0.3	32	97.56	0.23
0.03	0.45	11	100	0.27
0.03	0.45	32	97.56	0.17
0.03	0.5	8	95.12	0.32
0.04	0.15	33	97.56	0.21
0.04	0.2	6	97.56	0.51
0.04	0.2	33	97.56	0.23
0.04	0.2	36	95.12	0.17
0.04	0.25	6	97.56	0.59
0.05	0.05	36	100	0.18
0.05	0.1	23	97.56	0.3
0.05	0.2	5	100	0.59
0.05	0.2	6	97.56	0.59
0.05	0.2	23	97.56	0.26
0.06	0.45	34	97.56	0.19
0.06	0.45	36	95.12	0.2
0.06	0.5	5	100	0.49
0.06	0.5	23	95.12	0.29
0.06	0.5	34	97.56	0.18
0.07	0.05	23	95.12	0.27
0.07	0.2	5	97.56	0.68
0.07	0.2	23	95.12	0.35

Table 4: Random Point Anomalies Sample Output

SAMPLE Sensor Failure				
R^2 Tolerance	**Slope**	**Window Size**	**Accuracy**	**Time**
0.02	0.05	31	97.67	0.11
0.02	0.1	31	97.67	0.11
0.02	0.15	31	97.67	0.11
0.02	0.2	31	97.67	0.11
0.02	0.25	31	97.67	0.12
0.02	0.3	31	97.67	0.12
0.02	0.35	31	97.67	0.11
0.02	0.4	31	97.67	0.11
0.02	0.45	31	97.67	0.13
0.02	0.5	31	97.67	0.11

Table 5: Sensor Failure Sample Output

SAMPLE Linear Drift				
R^2 Tolerance	Slope	Window Size	Accuracy	Time
0.01	0.2	37	99.04	0.21
0.01	0.25	37	99.04	0.23
0.02	0.05	34	100	0.2
0.02	0.05	35	98.08	0.25
0.02	0.1	34	100	0.3
0.02	0.1	35	98.08	0.29
0.02	0.15	34	100	0.27
0.03	0.25	33	96.15	0.23
0.03	0.25	34	100	0.25
0.03	0.25	35	98.08	0.17
0.03	0.3	33	96.15	0.29
0.03	0.3	34	100	0.23
0.04	0.2	27	100	0.21
0.04	0.2	28	96.15	0.25
0.04	0.2	30	97.12	0.24
0.04	0.2	33	96.15	0.23
0.04	0.2	34	100	0.26
0.05	0.05	27	100	0.23
0.05	0.1	21	99.04	0.34
0.05	0.1	27	100	0.29
0.05	0.15	21	99.04	0.29
0.05	0.15	27	100	0.29
0.06	0.15	20	99.04	0.41
0.06	0.15	27	100	0.24
0.06	0.2	19	100	0.56
0.06	0.2	20	99.04	0.46
0.06	0.2	27	100	0.43
0.07	0.35	24	98.08	0.33
0.07	0.35	27	100	0.28
0.07	0.4	19	97.12	0.39
0.07	0.4	20	99.04	0.32
0.07	0.4	23	99.04	0.27
0.08	0.1	24	98.08	0.25
0.08	0.15	19	97.12	0.4
0.08	0.15	24	98.08	0.43
0.09	0.05	22	97.12	0.35
0.09	0.05	24	98.08	0.43
0.09	0.1	22	97.12	0.28

Table 6: Linear Drift Sample Output

SAMPLE 5% Down Deviation				
R^2 Tolerance	Slope	Window Size	Accuracy	Time
0.01	0.05	13	100.00	0.32
0.01	0.05	17	100.00	0.21
0.01	0.1	13	100.00	0.22
0.01	0.1	17	100.00	0.18
0.01	0.15	13	100.00	0.21
0.01	0.15	17	100.00	0.19
0.01	0.2	13	100.00	0.21
0.01	0.2	17	100.00	0.19
0.01	0.25	13	100.00	0.21
0.01	0.25	17	100.00	0.18
0.01	0.3	13	100.00	0.19
0.01	0.3	17	100.00	0.17
0.01	0.35	13	100.00	0.22
0.01	0.35	17	100.00	0.19
0.01	0.4	13	100.00	0.23
0.02	0.25	5	97.56	0.49
0.02	0.3	5	95.12	0.54
0.02	0.35	5	95.12	0.60
0.02	0.4	5	95.12	0.65
0.02	0.45	5	95.12	1.00
0.02	0.5	5	95.12	0.64
0.03	0.05	5	95.12	0.63
0.03	0.05	6	97.56	0.51
0.03	0.1	5	95.12	0.81
0.03	0.1	6	97.56	0.59
0.03	0.15	5	95.12	0.92
0.03	0.15	6	97.56	0.70
0.03	0.2	5	95.12	0.70
0.03	0.2	6	100.00	0.64
0.03	0.25	5	95.12	0.71
0.03	0.25	6	97.56	0.60

Table 7: 5% Down Deviation Sample Output

SAMPLE 10% Down Deviation				
R^2 Tolerance	Slope	Window Size	Accuracy	Time
0.01	0.15	9	97.06	0.41
0.01	0.15	13	100.00	0.38
0.01	0.15	38	97.06	0.11
0.01	0.2	9	97.06	0.43
0.01	0.2	13	100.00	0.31
0.01	0.2	38	97.06	0.09
0.01	0.25	9	97.06	0.40
0.01	0.25	13	97.06	0.30
0.01	0.25	38	97.06	0.11
0.01	0.3	9	97.06	0.39
0.01	0.3	13	97.06	0.30
0.01	0.5	9	97.06	0.33
0.01	0.5	13	97.06	0.30
0.01	0.5	38	97.06	0.09
0.02	0.05	5	97.06	0.75
0.02	0.05	6	97.06	0.47
0.02	0.1	5	97.06	0.83
0.02	0.1	6	97.06	0.64
0.02	0.15	5	97.06	0.57
0.02	0.15	6	97.06	0.62
0.02	0.2	5	97.06	0.61
0.02	0.2	6	97.06	0.66
0.02	0.25	5	97.06	0.56
0.02	0.25	6	97.06	0.56
0.02	0.3	5	97.06	0.63
0.02	0.3	6	97.06	0.61
0.02	0.35	5	97.06	0.70
0.02	0.35	6	97.06	0.57
0.02	0.4	5	97.06	0.72
0.02	0.4	6	97.06	0.64
0.02	0.45	5	97.06	0.61
0.02	0.45	6	97.06	0.54
0.02	0.5	5	97.06	0.54
0.02	0.5	6	97.06	0.49

Table 8: 10% Down Deviation Sample Output

SAMPLE Erratic Failure				
R^2 Tolerance	**Slope**	**Window Size**	**Accuracy**	**Time**
0.01	0.4	12	98.59	0.58
0.01	0.4	13	98.59	0.59
0.01	0.45	5	95.77	1.39
0.01	0.45	7	100.00	1.04
0.01	0.45	12	98.59	0.52
0.01	0.45	13	98.59	0.47
0.01	0.5	5	95.77	1.50
0.01	0.5	7	100.00	1.03
0.01	0.5	12	98.59	0.46
0.01	0.5	13	98.59	0.54
0.02	0.05	5	98.59	1.34
0.02	0.05	6	100.00	1.12
0.02	0.05	7	100.00	1.14
0.02	0.1	5	98.59	1.56
0.02	0.1	6	98.59	0.85
0.02	0.1	7	100.00	1.18
0.02	0.15	5	98.59	1.10
0.02	0.15	6	100.00	0.96
0.02	0.15	7	100.00	0.78
0.02	0.2	5	98.59	1.45
0.02	0.2	6	100.00	0.97
0.02	0.2	7	100.00	1.06
0.02	0.25	5	98.59	1.67
0.02	0.25	6	100.00	1.32
0.02	0.25	7	100.00	1.29

Table 9: Erratic Failure Sample Output

C UNUSUAL ANOMALY TEST CASES

SAMPLE 10% Up Deviation				
R^2 Tolerance	Slope	Window Size	Accuracy	Time
0.01	0.05	25	97.06	0.13
0.01	0.1	7	97.06	0.51
0.01	0.1	25	97.06	0.13
0.01	0.15	25	97.06	0.08
0.01	0.2	25	97.06	0.10
0.01	0.25	25	97.06	0.06
0.01	0.3	25	97.06	0.07
0.01	0.35	25	97.06	0.06
0.01	0.4	25	97.06	0.06
0.01	0.45	25	97.06	0.08
0.01	0.5	25	97.06	0.07

Table 10: 10% Up Deviation Sample Output

SAMPLE 5% Up Deviation				
R^2 Tolerance	**Slope**	**Window Size**	**Accuracy**	**Time**
0.01	0.05	8	100.00	0.32
0.01	0.1	8	100.00	0.31
0.01	0.2	8	100.00	0.30
0.01	0.3	8	100.00	0.30
0.01	0.35	8	100.00	0.30
0.01	0.4	8	100.00	0.39
0.01	0.45	8	100.00	0.39
0.02	0.05	8	100.00	0.36
0.02	0.1	8	100.00	0.29
0.02	0.15	8	100.00	0.33
0.02	0.2	8	100.00	0.45
0.02	0.25	8	100.00	0.42
0.02	0.35	8	100.00	0.27
0.02	0.45	8	100.00	0.37
0.02	0.5	8	100.00	0.40
0.03	0.1	8	100.00	0.34
0.03	0.15	8	100.00	0.36
0.03	0.2	8	100.00	0.40
0.03	0.25	8	100.00	0.39
0.03	0.3	8	100.00	0.32
0.03	0.4	8	100.00	0.40
0.03	0.45	8	100.00	0.36
0.03	0.5	8	100.00	0.38
0.04	0.1	8	100.00	0.40
0.04	0.2	8	100.00	0.37
0.04	0.35	8	100.00	0.49
0.04	0.45	8	100.00	0.48
0.06	0.05	5	97.06	0.59
0.06	0.1	5	97.06	0.73
0.06	0.15	5	97.06	0.56
0.06	0.2	5	97.06	0.69
0.06	0.25	5	97.06	0.74
0.06	0.3	5	97.06	0.50
0.06	0.35	5	97.06	0.69
0.06	0.4	5	97.06	0.68
0.06	0.45	5	97.06	0.65
0.06	0.5	5	97.06	0.68

Table 11: 5% Up Deviation Sample Output

SAMPLE Exponential Drift				
R^2 Tolerance	**Slope**	**Window Size**	**Accuracy**	**Time**
-	-	-	-	-

Table 12: Exponential Drift Sample Output

D RANDAP1000 ANOMALY COUNT SCRIPTS

D.1 Functions for Script

```python
# This program will house all the functions used for this particular script.

import pandas as pd
import numpy as np
from scipy import stats
from sklearn.cluster import KMeans


# First functions will be to create the dataframes
def create_dataframe(filename):
    df = pd.read_excel(filename)
    df.head()
    return df


def clean_data_columns(df, col_list_init, col_list_fin):
    if len(col_list_init) == 0:
        return col_list_fin

    # The following code should take care of the cases of specific variables as well as extremely
    # variable quantities or 0.
    # The column name cases are for known issues with columns in the dataset or known irrelevant
    # data.
    if min(df.loc[:, col_list_init[0]]) < 0 or max(df.loc[:, col_list_init[0]]) == 0 or min(
            df.loc[:, col_list_init[0]]) == max(df.loc[:, col_list_init[0]]):
        return clean_data_columns(df, col_list_init[1:], col_list_fin)

    elif str(col_list_init[0]) == 't' or str(col_list_init[0]) == 'Error':
        return clean_data_columns(df, col_list_init[1:], col_list_fin)

    elif str(col_list_init[0]) == 'TIME' or str(col_list_init[0]) == 'HTR':
        return clean_data_columns(df, col_list_init[1:], col_list_fin)

    else:
        col_list_fin.append(col_list_init[0])
        return clean_data_columns(df, col_list_init[1:], col_list_fin)


# The following will count the number anomalies inserted to the AP1000 datasets.
# This function ONLY works with these datasets.
def count_true_anomalies(df):
    anom_count = 0
    for i in range(df.shape[0]):
        if not df.loc[i, 'Error']:
            anom_count = anom_count + 1

    return anom_count


# Create Windows which will be evaluated with the linear regression.
def create_windows(df, col_name, win_size, rows_to_skip):
    # Function creates windows of designated size from a given dataframe's column.
    # This code has been modified to remove the first couple rows from the data to allow for some
    # stabilization.
    win_num = int(np.floor((df.shape[0] - rows_to_skip) / win_size))
    win_array = np.zeros((win_num, win_size))
    counter = rows_to_skip - 1
    for i in range(win_num):
        for j in range(win_size):
            win_array[i, j] = df.loc[counter, col_name]
            counter = counter + 1

    return win_array


# Next function is to create the linear regression and pull the r^2 value (should just return
# this value).
def find_r_square_slope(window):
    xs = np.array(range(len(window)))  # Create x-values from indexes of values in window
    ys = np.array(window)  # Convert window to numpy array
    res = stats.linregress(xs, ys)

    return res.rvalue ** 2, res.slope


# Last step (and the longest step) is to run a k-means cluster method.
# Will only be using a k=2 cluster since the data should be well-separated.
# Output from this function should only be the anomaly count due to the clustering (and maybe a
# pretty graph...).
def k_means_analysis(window, k):
    km = KMeans(n_clusters=k, n_init=10)
    xs = list(range(len(window)))
    xs_max = max(xs)

    # Normalize x-values utilizing 2*max(X)
    xs_norm = [x / (xs_max * 2) for x in xs]

    # Need to normalize y-values
    normalize_ys = lambda x: (x - min(window)) / (max(window) - min(window))
    window_norm = list(map(normalize_ys, window))

    data_frame = pd.DataFrame(list(zip(xs_norm, window_norm)), columns=['xs', 'ys'])

    y_pred = km.fit_predict(data_frame[['xs', 'ys']])
    data_frame['cluster'] = y_pred

    # The following code to create plots is mostly to make sure everything is running correctly
    group0 = []
    group1 = []
    mask1 = data_frame['cluster'] == 0
    mask2 = data_frame['cluster'] == 1
    df1 = data_frame.loc[mask1, ['xs', 'ys']]
    df2 = data_frame.loc[mask2, ['xs', 'ys']]

    if not df1.empty:
        for i in range(int(df1.shape[0])):
            group0.append([df1['xs'].values[i], df1['ys'].values[i]])

    if not df2.empty:
        for i in range(int(df2.shape[0])):
            group1.append([df2['xs'].values[i], df2['ys'].values[i]])

    return min(len(group0), len(group1))


# Function reports a simple accuracy calculation.
# Function has been adjusted to deal with over-counting.
def calculate_accuracy(experimental, true):
    return 1 - abs(true - experimental) / true
```

D.2 Report Anomalies Script

```python
from LinFunctions import *
import time
import matplotlib.pyplot as plt
import os

plt.style.use('seaborn-v0_8-notebook')

print('Script Starting... \n')

# Specify test ranges:
r_squared_range = [0.01 * 100, 0.10 * 100]  # r^2 value ranging from 0.01 to 0.10
slope_max = 5 * 1000

# Get File names of Excel files (for multiple anomaly sets)
file_names = []
for root, dirs, files in os.walk(
        r'C:\Users\POHLLM\OneDrive - Idaho National Laboratory\Documents\Anomaly_Detection\LinReg'
        r'\RandAP1000_Datasets'):
    for file in files:
        if file.endswith('.xlsx'):
            file_names.append(file)

# Create directory for outputs.
for excel_file in file_names:
    data_name = os.path.splitext(excel_file)[0]
    print(f'Starting {data_name}.')
    os.makedirs(os.path.join(r'C:\Users\POHLLM\OneDrive - Idaho National Laboratory\Documents'
                             r'\Anomaly_Detection\LinReg',
                             data_name),
                exist_ok=True)  # Create new directories to more cleanly store generated plots

    # Create and clean data
    # Count number of anomalies. This function only works with the RandAP1000 files.
    df = create_dataframe('RandAP1000_Datasets\\' + excel_file)
    true_anomalies = count_true_anomalies(df)
    column_list = clean_data_columns(df, list(df.columns), [])
    df = df[df.columns.intersection(column_list)]

    window_size_range = [5, int(np.floor(
        df.shape[0] / 4))]  # This is generalized to explore a range based on the size of the set.

    # Initialize anomaly count
    # Anomalies will be reported in the form:
    # r_sqr_tol, win_size, column, window_index
    # The last entry will ideally be used to find which value in the specified window.
    anomaly_list = []
    valid_configs = []

    print('Beginning calculations... \n')

    # First, vary r^2 value.
    for r_sqr_tol in range(int(r_squared_range[0]), int(r_squared_range[1])):
        r_sqr_tol = r_sqr_tol / 100

        acc_list = []
        win_list = []

        # We then vary the slope tolerance over a wide range.
        for slope_mag_max in np.arange(50, slope_max / 10 + 1, 50):
            slope_mag_max = slope_mag_max / 1000
            slope_range = [-slope_mag_max, slope_mag_max]

            print(f'Beginning {slope_mag_max}...')

            # Then vary window size by steps of 10.
            for win_size in range(window_size_range[0], window_size_range[1] + 1):

                # Initialize anomaly count
                anomaly_count = 0

                # Create timer for reporting speed
                t = time.time()

                # Next loop will take loop through all columns of the imported data.
                for col in column_list:
                    windows = create_windows(df, col, win_size, 3)
                    for win_index in range(windows.shape[0]):

                        # This should look at the specific window by allowing all columns.
                        # This next statement takes care of complete system failure
                        if max(windows[win_index, :]) == 0:
                            anomaly_list.append((r_sqr_tol, slope_mag_max, win_size, col,
                                                 win_index, 'Failure'))
                            anomaly_count = anomaly_count + len(windows[win_index, :])
                        else:
                            # Because the large values of Flow_A create a large slope regardless
                            # of deviation, we will normalize the windows to specify the window
                            # size for every case.
                            window = windows[win_index, :]

                            normalize_ys = lambda y: (y / (max(window)))
                            window_norm = list(map(normalize_ys, window))

                            r_sqr_test, slope_test = find_r_square_slope(window_norm)

                        # Statement looks for whether the data can be well-represented by a linear
                        # interpolation. Because of the definition of r^2, a LOW r^2 value is
                        # preferred with a small slope (only for the case of steady-state).

                        # If the data cannot be, a k-means analysis will be used to separate the
                        # anomalies. If the data can be represented linearly, but has a slope
                        # deviating too far from 0, it will be documented as a drift anomaly.
                        if (slope_range[0] < slope_test < slope_range[1] and r_sqr_test < r_sqr_tol
                            and max(
                                windows[win_index, :]) != min(windows[win_index, :])) or (
                                max(windows[win_index, :]) != 0 and min(windows[win_index, :])
                                == 0):
                            anom_found = k_means_analysis(windows[win_index, :], 2)
                            anomaly_count = anomaly_count + anom_found
                            anomaly_list.append((r_sqr_tol, slope_mag_max, win_size, col,
                                                 win_index, 'Point'))

                        # Small adjustment made in this line. Will only report drift anomalies if
                        # the data is already within the tolerance of the r^2 value. Otherwise,
                        # we risk way overcounting.
                        elif not (slope_range[0] < slope_test < slope_range[1]) and r_sqr_test > \
                                r_sqr_tol:
                            anomaly_list.append((r_sqr_tol, slope_mag_max, win_size, col,
                                                 win_index, 'Drift'))
                            anomaly_count = anomaly_count + len(windows[win_index, :])

                elapsed_time = time.time() - t

                accuracy = calculate_accuracy(anomaly_count, true_anomalies)

                acc_list.append(accuracy * 100)
                win_list.append(win_size)
                if accuracy >= 0.95:
                    valid_configs.append([r_sqr_tol, slope_mag_max, win_size, accuracy * 100,
                                          elapsed_time])

        print(f'R^2 of {r_sqr_tol}: complete \n')
        print(f'Max accuracy found: {max(acc_list)} \n')

    print(f'Creating and exporting dataframes...')
    a_con_df = pd.DataFrame(valid_configs, columns=['rvalue', 'slopemagnitude', 'windowsize',
                                                    'accuracy', 'time'])
    a_con_df.to_csv(f'{data_name}/{data_name}_LinWinK_ValidConfigs.csv')

    anom_df = pd.DataFrame(anomaly_list, columns=['rvalue', 'slopemagnitude', 'windowsize',
                                                  'column', 'windowindex',
                                                  'type'])
    anom_df.to_csv(f'{data_name}/{data_name}_LinWinK_Anomalies_Found.csv')
    print(f'Data exported. \n')

print('Script complete.')
```

E RAVEN SCRIPTS

E.1 XML File

```
<?xml version="1.0" ?>
<Simulation verbosity="silent">
```

```xml
    <RunInfo>
        <WorkingDir>raven_runs</WorkingDir>
        <Sequence>generate_data,stats</Sequence>
        <batchSize>1</batchSize>
    </RunInfo>

    <Steps>
        <MultiRun name="generate_data">
            <Sampler class="Samplers" type="MonteCarlo">my_mc</Sampler>
            <Input class="DataObjects" type="PointSet">placeholder</Input>
            <Model class="Models" type="ExternalModel">LinWinK_Raven</Model>
            <Output class="DataObjects" type="PointSet">results</Output>
        </MultiRun>
        <PostProcess name='stats'>
            <Input class='DataObjects' type='PointSet'>results</Input>
            <Model class='Models' type='PostProcessor'>stats</Model>
            <Output class='DataObjects' type='PointSet'>results2</Output>
            <Output class="OutStreams" type="Print">stats</Output>
        </PostProcess>
    </Steps>

    <Models>
        <ExternalModel
                ModuleToLoad="C:\Users\POHLLM\OneDrive - Idaho National Laboratory\Documents\
                Anomaly_Detection\RAVEN Sensitivity\LinWinK_Raven.py"
                name="LinWinK_Raven" subType="">
            <inputs>r_sqr_tol, slope_max, win_size</inputs>
            <outputs>accuracy, time</outputs>
        </ExternalModel>
        <PostProcessor name='stats' subType='BasicStatistics'>
            <!--      <dataset>True</dataset>-->
            <NormalizedSensitivity prefix='nsense'>
                <targets>accuracy, time</targets>
                <features>r_sqr_tol, slope_max, win_size</features>
            </NormalizedSensitivity>
        </PostProcessor>
    </Models>

    <Samplers>
        <MonteCarlo name="my_mc">
            <samplerInit>
                <limit>500</limit>
            </samplerInit>
            <variable name="r_sqr_tol">
                <distribution>r_sqr_dist</distribution>
            </variable>
            <variable name="slope_max">
                <distribution>slope_dist</distribution>
            </variable>
            <variable name="win_size">
                <distribution>window_size_dist</distribution>
            </variable>
        </MonteCarlo>
    </Samplers>

    <Distributions>
        <Uniform name="r_sqr_dist">
            <lowerBound>0.01</lowerBound>
            <upperBound>0.10</upperBound>
        </Uniform>
        <Uniform name="slope_dist">
            <lowerBound>0.05</lowerBound>
            <upperBound>0.5</upperBound>
        </Uniform>
        <UniformDiscrete name="window_size_dist">
            <lowerBound>5</lowerBound>
            <upperBound>20</upperBound>
            <strategy>orderedWithReplacement</strategy>
        </UniformDiscrete>
    </Distributions>

    <DataObjects>
        <PointSet name="placeholder">
            <Input>r_sqr_tol, slope_max, win_size</Input>
        </PointSet>
        <PointSet name="results">
            <Input>r_sqr_tol, slope_max, win_size</Input>
            <Output>accuracy, time</Output>
        </PointSet>
        <PointSet name="results2">
            <Output>nsense_accuracy_r_sqr_tol, nsense_accuracy_slope_max, nsense_accuracy_win_size,
                nsense_time_r_sqr_tol, nsense_time_slope_max, nsense_time_win_size
            </Output>
        </PointSet>
    </DataObjects>

    <OutStreams>
        <Print name="stats">
            <type>csv</type>
            <source>results2</source>
        </Print>
    </OutStreams>

</Simulation>
```

E.2 Adjusted Python Script

```python
def run(self, Input):
    # This program will house all of the functions used for this particular script.
    # Scripts will be recycled (if possible) from SimpleWindows.py project
    import pandas as pd
    from scipy import stats
    from sklearn.cluster import KMeans
    import random
    import time
    import numpy as np

    # First functions will be to create the dataframes
    def create_dataframe(filename):
        df = pd.read_excel(filename)
        df.head()
        return df

    def clean_data_columns(df, col_list_init, col_list_fin):
        if len(col_list_init) == 0:
            return col_list_fin

        # The following code should take care of the cases of specific variables as well as
        # extremely variable quantities or 0. A new condition was added to ideally remove
        # any empty columns.

        if str(col_list_init[0]) == 'TIME' or str(col_list_init[0]) == 'HTR':
            return clean_data_columns(df, col_list_init[1:], col_list_fin)

        elif min(df.loc[:, col_list_init[0]]) < 0 or max(df.loc[:, col_list_init[0]]) == 0 or min(
                df.loc[:, col_list_init[0]]) == max(df.loc[:, col_list_init[0]]):
            return clean_data_columns(df, col_list_init[1:], col_list_fin)

        else:
            col_list_fin.append(col_list_init[0])
            return clean_data_columns(df, col_list_init[1:], col_list_fin)

    # Create Windows which will be evaluated with the linear regression
    def create_windows(df, col_name, win_size, rows_to_skip):
        # Function creates windows of designated size from a given dataframe's column.
        # This code has been modified to remove the first couple rows from the data to allow for
        # some stabilization.
        win_num = int(np.floor((df.shape[0] - rows_to_skip) / win_size))
        win_array = np.zeros((win_num, win_size))
        counter = 0
        for i in range(win_num):
            for j in range(win_size):
                win_array[i, j] = df.loc[counter, col_name]
                counter = counter + 1

        return win_array

    # Next function is to create the linear regression and pull the r^2 value (should just return
    # this value)
    def find_r_square_slope(window):
        xs = np.array(range(len(window)))  # Create x-values from indexes of values in window
        ys = np.array(window)  # Convert window to numpy array
        res = stats.linregress(xs, ys)

        return res.rvalue ** 2, res.slope

    # Last step (and the longest step) is to run a k-means cluster method.
    # Will only be using a k=2 cluster since the data should be well-separated.
    # Output from this function should only be the anomaly count due to the clustering.
    def k_means_analysis(window, k):
        km = KMeans(n_clusters=k, n_init=10)
        xs = list(range(len(window)))
        xs_max = max(xs)

        # Normalize x-values utilizing 2*max(X)
        xs_norm = [x / (xs_max * 2) for x in xs]

        # Need to normalize y-values
        normalize_ys = lambda x: (x - min(window)) / (max(window) - min(window))
        window_norm = list(map(normalize_ys, window))

        data_frame = pd.DataFrame(list(zip(xs_norm, window_norm)), columns=['xs', 'ys'])

        y_pred = km.fit_predict(data_frame[['xs', 'ys']])
        data_frame['cluster'] = y_pred

        group0 = []
        group1 = []
        mask1 = data_frame['cluster'] == 0
        mask2 = data_frame['cluster'] == 1
        df1 = data_frame.loc[mask1, ['xs', 'ys']]
        df2 = data_frame.loc[mask2, ['xs', 'ys']]

        if not df1.empty:
            for i in range(int(df1.shape[0])):
                group0.append([df1['xs'].values[i], df1['ys'].values[i]])

        if not df2.empty:
            for i in range(int(df2.shape[0])):
                group1.append([df2['xs'].values[i], df2['ys'].values[i]])

        return min(len(group0), len(group1))

    def calculate_accuracy(experimental, true):
        return 1 - abs(true - experimental) / true

    def randomize_dataframe(dataframe, pertrange, max_in_row):
        df = dataframe
        all_columns = list(df.columns)
        n_columns = random.randrange(1, len(all_columns))
        if n_columns > df.shape[1]:  # Might need to check this
            return

        anom_count_true = 0

        # first step is to randomly select the columns
        if len(all_columns) == n_columns:
            columns = all_columns
        else:
            columns = []
            indexes = list(range(len(all_columns)))
            random.shuffle(indexes)  # Create permutation of indexes
            for i in indexes:
                columns.append(all_columns[i])  # This should choose random columns
            columns = columns[0:n_columns]

        # Now to randomly perturb the rows
        for col in columns:
            num_in_row = random.randrange(1, max_in_row)
            anom_count_true = anom_count_true + num_in_row
            for i in range(num_in_row):
                row = random.randrange(0, df.shape[0])
                if df.loc[row, col] != 'NaN':
                    # Notation looks weird, but that's because randrange is not inclusive on the
                    # upper bound.
                    df.loc[row, col] = df.loc[row, col] * (
                            1 + (-1) ** random.randrange(1, 3) *
                            random.randrange(pertrange[0], pertrange[1]) / 100)
                    if df.loc[row, col] == 0:
                        anom_count_true = anom_count_true - 1
                else:
                    anom_count_true = anom_count_true - 1

        return [df, anom_count_true]

    # This script will couple with RAVEN to perform a sensitivity analysis on r^2, slope_max, and
    # window_size.

    r_sqr_tol = self.r_sqr_tol
    slope_max = self.slope_max
    win_size = int(self.win_size)

    # Create and clean data
    df = create_dataframe(
        r'C:\Users\POHLLM\OneDrive - Idaho National Laboratory\Documents\Anomaly_Detection\
        RAVEN Sensitivity\SteadyState.xlsx')
    column_list = clean_data_columns(df, list(df.columns), [])
    df = df[df.columns.intersection(column_list)]

    # Randomize Dataframe, count anomalies
    max_in_row = 10
    pert_range = [5, 100]  # Allows for max perturbations between 5% and 100%
    [df, true_anomalies] = randomize_dataframe(df, pert_range, max_in_row)

    # We then vary the slope tolerance over a wide range.
    slope_range = [-slope_max, slope_max]

    # Then vary window size by steps of 10.

    anomaly_count = 0

    # Create timer for reporting speed
    t = time.time()

    # Next loop will take loop through all columns of the imported data.
    for col in column_list:
        nan_indices = df.index[df[col].isna()].tolist()

        if nan_indices:
            df = df.loc[0:min(nan_indices) - 1, :]

        number_of_elements = df.shape[0]

        # This new 'if' statement is necessary because of the varying column sizes.
        if int(np.floor(number_of_elements / 4)) >= win_size:

            windows = create_windows(df, col, win_size, 0)

            for win_index in range(windows.shape[0]):

                # This should look at the specific window by allowing all columns.
                # This next statement takes care of complete system failure
                if max(windows[win_index, :]) == 0:
                    anomaly_count = anomaly_count + len(windows[win_index, :])
                    slope_test = 0
                    r_sqr_test = 0
                else:
                    # Because the large values of Flow_A create a large slope regardless of
                    # deviation, we will normalize the windows to specify the window size
                    # for every case.
                    window = windows[win_index, :]

                    normalize_ys = lambda y: (y / (max(window)))
                    window_norm = list(map(normalize_ys, window))

                    r_sqr_test, slope_test = find_r_square_slope(window_norm)

                # Statement looks for whether the data can be well-represented by a linear
                # interpolation. Because of the definition of r^2, a LOW r^2 value is
                # preferred with a small slope (only for the case of steady-state).

                # If the data cannot be, a k-means analysis will be used to separate the anomalies.
                # If the data can be represented linearly, but has a slope deviating too far from
                # 0, it will be documented as a drift anomaly.
                if (slope_range[0] < slope_test < slope_range[1] and r_sqr_test < r_sqr_tol
                    and max(windows[win_index, :]) != min(windows[win_index, :])) or \
                        (max(windows[win_index, :]) != 0 and min(windows[win_index, :]) == 0):
                    anom_found = k_means_analysis(windows[win_index, :], 2)
                    anomaly_count = anomaly_count + anom_found

                # Small adjustment made in this line. Will only report drift anomalies if the data
                # is already within the tolerance of the r^2 value. Otherwise, we risk way
                # overcounting.
                elif not (slope_range[0] < slope_test < slope_range[1]) and r_sqr_test > r_sqr_tol:
                    anomaly_count = anomaly_count + len(windows[win_index, :])

        elapsed_time = time.time() - t
        accuracy = calculate_accuracy(anomaly_count, true_anomalies)

        self.accuracy = accuracy
        self.time = elapsed_time
```

REFERENCES

[1] V. Yadav, V. Agarwal, P. Jain, P. Ramuhalli, and X. Zhao, “State-of-technology and technical challenges in advanced sensors, instrumentation, and communication to support digital twin for nuclear energy application,” 02 2023.

[2] F. Dong, S. Chen, K. Demachi, M. Yoshikawa, A. Seki, and S. Takaya, “Attention-based time series analysis for data-driven anomaly detection in nuclear power plants,” *Nuclear Engineering and Design*, vol. 404, p. 112161, 2023.

[3] F. A. Mazarbhuiya and M. Shenify, “A mixed clustering approach for real-time anomaly detection,” *Applied Sciences*, vol. 13, no. 7, p. 4151, 2023.

[4] E. Jumin, N. Zaini, A. N. Ahmed, S. Abdullah, M. Ismail, M. Sherif, A. Sefelnasr, and A. El-Shafie, “Machine learning versus linear regression modelling approach for accurate ozone concentrations prediction,” *Engineering Applications of Computational Fluid Mechanics*, vol. 14, no. 1, pp. 713–725, 2020.

[5] T. Sauer, *Numerical Analysis*. Greg Tobin, 2006.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[7] N. University, "Coefficient of determination, r-squared," 2023.

[8] "K means clustering - introduction - geeksforgeeks," 05 2019.

[9] "Pctran: A pc-based simulator for the westinghouse ap1000," 2007.

[10] "Raven," 07 2023.

[11] T. Alade, "Tutorial: How to determine the optimal number of clusters for k-means clustering," 12 2019.