A DATA-DRIVEN APPROACH TO CUBESAT HEALTH MONITORING

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Serbinder Singh

June 2017

COMMITTEE MEMBERSHIP

TITLE:	A Data-Driven Approach to CubeSat Health Monitoring
AUTHOR:	Serbinder Singh
DATE SUBMITTED:	June 2017
COMMITTEE CHAIR:	John Bellardo, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	John Seng, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Lubomir Stanchev, Ph.D. Associate Professor of Computer Science

ABSTRACT

A Data-Driven Approach to CubeSat Health Monitoring

Serbinder Singh

Spacecraft health monitoring is essential to ensure that a spacecraft is operating properly and has no anomalies that could jeopardize its mission. Many of the current methods of monitoring system health are difficult to use as the complexity of spacecraft increase, and are in many cases impractical on CubeSat satellites which have strict size and resource limitations. To overcome these problems, new data-driven techniques such as Inductive Monitoring System (IMS), use data mining and machine learning on archived system telemetry to create models that characterize nominal system behavior. The models that IMS creates are in the form of clusters that capture the relationship between a set of sensors in time series data. Each of these clusters define a nominal operating state of the satellite and the range of sensor values that represent it. These characterizations can then be autonomously compared against real-time telemetry on-board the spacecraft to determine if the spacecraft is operating nominally.

This thesis presents an adaption of IMS to create a spacecraft health monitoring system for CubeSat missions developed by the PolySat lab. This system is integrated into PolySat's flight software and provides real time health monitoring of the spacecraft during its mission. Any anomalies detected are reported and further analysis can be done to determine the cause. The system can also be used for the analysis of archived events. The IMS algorithms used by the system were validated, and ground testing was done to determine the performance, reliability, and accuracy of the system. The system was successful in the detection and identification of known anomalies in archived flight telemetry from the IPEX mission. In addition, real-time monitoring performed on the satellite yielded great results that give us confidence in

the use of this system in all future missions.

ACKNOWLEDGMENTS

Thanks to:

- My advisor, John Bellardo, for guiding and helping me throughout this process. His input has been invaluable to the success of this thesis.
- David Iverson, for being a huge help throughout the research and validation phases of this thesis. As the original creator of the IMS system, his feedback was very important.
- Luc Bouchard, for giving his input on the design of the system.
- Christopher Gerdom, for help validating some of the results.
- My parents for supporting me throughout my life
- The entire Computer Science Department at Cal Poly.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

The role of satellites in the current technological age is immense. They provide essential services and modern conveniences that have become ingrained into society's fabric. Some of the many important applications of satellites include radio communications, GPS, weather monitoring, broadcast media, and scientific research. Thousands of satellites have been deployed in various orbits around Earth and beyond that either provide services or conduct experiments. Due to this big role that satellites play in our lives, it is important that proper care and maintenance is given to these systems so they remain reliable and function properly. As a result, satellite health monitoring, which involves verifying the proper functionality of all systems on board, is essential to ensure that a satellite is operating properly and has no anomalies that could jeopardize its mission.

There have been many advancements and improvements made throughout the years on the capabilities and functions of various satellites. The design of such systems that fulfill all the specifications make them extremely sophisticated and complex [11]. Unfortunately, the monitoring of such systems also becomes very complex as there are many sensor and component interactions that become hard to predict and classify as nominal through traditional techniques. There are currently many traditional methods of monitoring spacecraft that include parameter limit checking, model-based, and rule-based techniques that become difficult and cumbersome as the complexity of the spacecraft increases [10]. These challenges are exacerbated in CubeSat satellites where using extra downlink capacity for high resolution engineering telemetry is not feasible. In order to overcome some of the problems with traditional approaches to health monitoring, new data-driven techniques based on data-mining and machine learning have been developed to make this task much simpler and manageable.

These techniques create models that characterize nominal system behavior by looking at an archive of nominal system telemetry. These characterizations can then be autonomously compared against real-time telemetry on-board the spacecraft to determine if the spacecraft is operating nominally. This approach has many advantages over traditional approaches; the main ones being efficiency, simplicity, and adaptability. One such data-driven system is called Inductive Monitoring System (IMS) and has been successfully implemented in various applications.

IMS models the relationship between a set of sensors in time series data as clusters. It goes through the archived data and matches consistent or similar telemetry points into the same cluster. Each of these clusters end up defining a different nominal state of the spacecraft and the range of sensor values that represent it. The end result of the training portion is a knowledge base of clusters that can be compared against real-time telemetry to see if any readings fall outside the known good clusters. If there is a cluster match, then the system is most likely performing nominally; otherwise there may be an anomaly. IMS is beneficial in that it allows modeling the complex interactions between related parameters instead of looking at the parameters individually.

1.1 Contribution

The goal of this thesis is to use the research and work done on the Inductive Monitoring System and apply it to create a validated and flight ready system for use on CubeSat satellites. CubeSat's are miniature satellites that are primarily used for space research by many educational institutions and private firms [1]. PolySat is a club on the Cal Poly San Luis Obispo campus that designs, fabricates, and tests CubeSats. This particular health monitoring system will be integrated into PolySats flight soft-

ware to provide real-time health monitoring of the spacecraft during its mission. The system flags anomalies that may be occurring and reports them back to the ground for analysis. This paper covers the system design of this health monitoring module and how it communicates with different subsystems on the satellite. Performance and results of the health monitoring system are explored through ground testing. These tests include detection of known anomalies using archived flight telemetry from the IPEX mission, injecting failures in real-time, and running computation profiling tools to examine overall resource requirements. The benefits and limitations of using this system will be examined, and a recommendation will be made on how useful it is for use in other missions.

Chapter 2

BACKGROUND

The focus of this thesis is to create a validated and flight ready implementation of an autonomous spacecraft health monitoring system for use on CubeSat satellites developed by the PolySat lab. Specifically, the Inductive Monitoring System is used because of its simplicity, power, and success on similar applications. To give some background on the material this paper will cover, this section discusses data mining and machine learning, different spacecraft monitoring techniques including IMS, an overview of PolySat, and the design of the flight software in which this system is implemented.

2.1 Data Mining and Machine Learning

In this current age of computing, information in the form of raw data is being generated at a huge scale that has resulted in large data warehouses containing statistics ranging from what websites you visit, to detailed records of your spending. This trend of collecting large amounts of data also applies to the field of aeronautics and spacecraft. There are large archives of system telemetry that have been gathered by organizations such as NASA from many spacecraft missions. These data sets give valuable information on the state of various subsystems and sensors on-board different spacecraft during their mission [13]. This underlying data may contain patterns and relationships that are not visible through manual inspection alone. These patterns may reveal potentially important information that can be used to glean a better understanding of the underlying data.

The goal of data mining is to extract the implicit, previously unknown, and potentially useful information from the data [22]. This process of discovering useful patterns in data is usually semi or fully autonomous and the patterns that are mined are represented in a structure that can be easily examined and reasoned about. The data can also be used to make informed future decisions. This information may reveal strong patterns between certain data points that allow accurate predictions to be made on future data that matches these patterns. In the real world, a lot of data may be imperfect in that some portions may be missing or reveal no patterns at all. Algorithms used in data mining are robust enough to ignore imperfections and still find regularities that may be useful.

Machine learning provides the technical means of performing data mining [22]. In other words, it is the technique for finding, describing, and learning these structural patterns that can be found in the data. Programs can now use machine learning techniques to extract useful information from the data and train their system to learn from the patterns and relationships found. By doing this, you can make informed future decisions and perform a lot of autonomous functions. No longer does the program have to be explicitly programmed; instead, the behavior is learned from large sets of data. Machine learning is being extensively used in a wide array of fields ranging from facial recognition, to complex medical machines used to perform surgeries [20]. Data mining and machine learning is used in this thesis to generate complex models that model nominal system behavior in CubeSats and allow for the monitoring of future data.

2.1.1 Learning techniques

The field of statistics is very important in machine learning because it aids in the generation of the mathematical models that describe the patterns and variations in the data. The field of computer science also has two big roles in machine learning.

First is using efficient algorithms to process the data and generating or “training” the mathematical model that describes this data. Once the model is generated, its representation and algorithmic solution also needs to be efficient and provide results in a timely manner [5]. This highlights two important phases that are seen throughout machine learning, the training/learning phase where the models that describe the data are generated, and the monitoring/inference stage where those models are used to understand the data and make new predictions based off of it.

There are also different types of learning algorithms that are used to train the models. Which of these algorithms to use is usually determined by the type of data that is used. These algorithms generally fall in two main categories, Supervised and Unsupervised learning algorithms.

2.1.2 Supervised Learning

Currently, the large majority of machine learning applications use some form of supervised learning [4]. In supervised learning, labeled training data is used to learn the mapping function that maps the input variables to the output variables. This mapping function is approximated to then predict the output variables based on new input data. The learning is “supervised” in that for each input variable, we know what the output should be and the function will be optimized to learn this. There are many different supervised learning algorithms that can be used to create this approximation function. These can be grouped into classification and regression problems.

Classification problems are those which have a defined class or category that its output variable falls within. An example of this could be a binary classifier that predicts whether or not it will rain. The set of input variables or “features” that the classifier uses to create the mapping function could be various attributes of the day such as humidity, temperature, and pressure. The output in this example would fall into one of two categories, yes or no. The model will be trained with previous labeled data that contain both vectors of features and the output yes or no. After the system has been trained, it can then be used to predict future output based on new input. The classifiers in classification problems can have many categories that the input data can map to and this number usually depends on the specific problem [5].

Regression problems are different from classification in that the output variable no longer falls into a category; rather, it is a real number value such as dollars or weights [4]. These problems also take in labeled training data and use the input features and their output values to create a fitting function that maps the input to the output. It can then predict a value from this function based on new input data.

2.1.3 Unsupervised Learning

As discussed, supervised learning algorithms contained labeled training data where the input variables are mapped to an output. In unsupervised learning, the training data only contains the input, and there is no output for which we can create a fitting function. The goal for these algorithms is to discover regularities and patterns in the input space, model the structure and distribution in the data, and see if we can learn something useful from the data. This is known as density estimation in statistics [5].

One of the methods to do this density estimation is called clustering where the goal is to find clusters or groupings in which the input fall. For example one may want to cluster customers based on purchasing behavior. There could be many clusters that are formed from any given input and the points in each cluster are very similar to each other and different from points in another cluster. Once these clusters are formed, future data points can be compared against them to see if there are any matches. Clustering can be used for classification where each grouping of data defines a different class. It differs from supervised learning classification in that it decides these classes looking at unlabeled data, rather than having labeled data that specifies the class the point belongs to. Clustering is also good for anomaly detection where there is only one class that the data defines; nominal data. Once this data has been separated into clusters, it is easy to find anomalous points or outliers that do not fit any of the grouping that represent nominal behavior. There are several clustering algorithms such as K-means and density-based approaches that form these clusters in different ways. [4]. Clustering is the main machine learning technique that is utilized by the Inductive Monitoring System to provide its monitoring capabilities. We will discuss the algorithm in more detail later.

2.2 Spacecraft Health Monitoring

Spacecraft health monitoring is essential to ensure that a spacecraft is operating properly and has no anomalies that could jeopardize its mission. This monitoring involves a host of people, including mission controllers and systems engineers, who monitor the down-linked data and analyze it. As mentioned earlier, the increased capabilities of modern satellites has led to incredibly sophisticated systems with complex interactions between hardware components and software. As a result, the monitoring of the system's health also becomes very complex. We will take a look at traditional methods of spacecraft monitoring, and then present a more practical data-driven technique that can be used alongside existing systems to provide accurate and valuable decision support for the monitoring of a satellite.

One method of health tracking is parameter limit checking where a reference table of nominal sensor values is created for all the sensors across the system. These values will specify a range where such sensors can be deemed healthy. If a value doesn't fall within this range, then that particular component may have an anomaly [11].

This method of health monitoring is very inefficient and time consuming because as the number of components increases, the generation of this reference table becomes extremely hard. It is very difficult to correctly determine what would constitute a healthy sensor value. Also, multiple reference tables would have to be made for each of the satellites different operational modes due to different component interactions in each case. Another drawback of such an approach is that it only considers individual parameter ranges when making its decision, and can't model complex interactions that may involve several concurrent parameters in the operating context.

Since we want more functions on the satellite to be autonomous and not need human interaction to determine whether the state of a satellite is nominal, we need to make the monitoring more autonomous. One common approach to anomaly and fault detection within satellite systems is hardware redundancy. This method makes direct comparisons from multiple identical sensors and uses a voting system to try to identify a faulty sensor. If one sensor provides faulty results, we can identify which one it is. This method requires very little computation, however can be expensive and space-limiting [7]. It is also not practical for CubeSats because redundant hardware is expensive in both size and power requirements.

2.3 Inductive Health Monitoring System

In order to address the challenge of monitoring the increasingly complex component interactions of modern spacecraft and other aerospace related systems, new data-driven techniques have been developed that provide more advanced system health monitoring. These techniques have been made possible to use by the abundance of archived system telemetry that has been collected over the years for several different spacecraft and applications. These systems can provide valuable decision support for the people responsible for monitoring the system [13]. These data driven techniques try to characterize nominal system behavior models by analyzing archived operational data fed to it [10]. Access to extensive nominal operational data allows for the creation of complex characterizations of component interactions that are only discovered by using data mining and machine learning techniques such as clustering. These characterizations can than be used for health monitoring by comparing them to real-time telemetry. If the system is performing nominally, the telemetry will fall under one of the models. If not, than this may indicate a potential anomaly or fault in the system.

One benefit of such data-driven approaches over existing ones is that a deep knowledge of the system that needs to be monitored is not required. There is no need to figure out what constitutes a healthy system and how the internal components interact in order to create a model that represents this. Only archived data is needed to determine the nominal ranges of operation. Another benefit is that special analysis is not required to determine the relationships among sensors because these relationships and patterns are found from the data itself. This is very beneficial because it becomes extremely hard to model these relationships as the complexity of the spacecraft increases. Data driven techniques can also work in high dimensional spaces and multiple sensors can be examined concurrently rather than individual parameter checking. Overall, data driven techniques provide many benefits that makes the job of health monitoring simpler and more efficient. One such data driven system is the Inductive Monitoring System (IMS).

IMS was developed by an engineer at NASA, David Iverson, and is a data-driven health monitoring technique that models the relationship between a set of sensors in time-series data as clusters. IMS uses vectors as a data structure that holds the values of several related system parameters for a specific time. It goes through the archived data, forms these vectors, and groups vectors with similar or consistent values in the same cluster. Therefore, each cluster has a set of vectors that defines a different characterization or nominal state that the system can be in and the sensor ranges that represent it. This is beneficial because it allows us to model interactions between related parameters instead of looking at each one individually. The end result is a knowledge base of many clusters that define a model of the nominal states of the system. This can be used for real time monitoring or analysis of archived events [13]. Since the training data used only has one class, nominal, a new vector that falls into any of the clusters will be accepted. Because this application of clustering isn't classification, it does not matter which cluster the vector falls within, as long as it is in one. If the point is anomalous, it will be an outlier in the data space and not belong to any cluster.

For monitoring, new spacecraft data can be input and compared against the model generated by IMS and a deviation value can be calculated that defines how close or far the current system behavior is from nominal. A high deviation value could signify a malfunction in the system and alert mission controllers to perform a closer inspection of the data. It is important to note that IMS does not solve or pinpoint the exact cause of anomalies on the spacecraft or application. It can only detects anomalies and gives details on which sensors or features may be causing the issue so that it is easier to find the problem.

IMS as a tool for anomaly detection can be thought of consisting of two main phases, learning and monitoring. The learning modules takes as input archived data and some learning parameters, and then uses machine learning in the form of clustering to create a model of the nominal operating states for that particular system. The output of the learning phase produces a knowledge base of this model consisting of many clusters that each represent a different nominal state of the spacecraft. The monitoring phase uses the knowledge base that was created by the learning phase to monitor any new input data that is given to it. The output of the monitoring phase is a deviation score of how well the inputted data matches the nominal clusters in the knowledge base. A high level diagram of these two phases in IMS is shown in figure 2.1. We will now take a deeper look into how these two phases work and how the algorithms work.

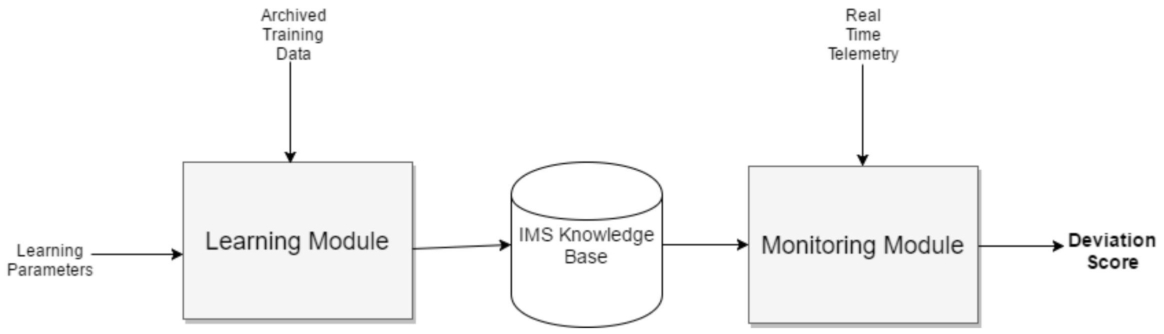
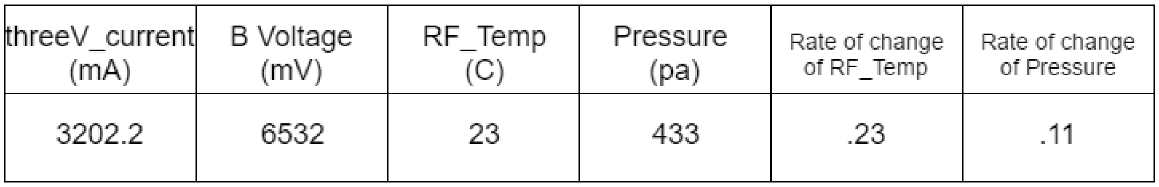


Figure 2.1: High-level overview Inductive Health monitoring systems

2.3.1 IMS Learning

The first step of IMS and other data-driven health monitoring techniques is the Training/Learning phase. IMS uses an unsupervised clustering algorithm for training that will capture the patterns and regularities from the data and output a model in the form of multiple clusters of similar points [11]. In this step, nominal archived operational data is used to generate the models that characterize the different nominal clusters. It is important that this data be free of any errors and checked for accuracy because if not, incorrect system behavior may be incorporated into the model and provide bad results. If data with anomalies is included within the dataset, then if the same undesired behavior occurs, the system will think it nominal. It is also important to ensure that the training data is extensive enough to provide the model the ability to capture all possible interactions and behavior from the set of sensors. If there is not enough training data, then normal system behavior that may have not been modeled by the training data may incorrectly define the behavior of a healthy system as anomalous.

In the case of IMS, a vector of system parameter values is used as the data structure that holds the information necessary to create the model and is shown in Figure 2.2. Mission controllers would want to pick the parameter values they are most interested in monitoring and place them in the vector. We will call these parameters or sensors chosen as the vectors “features”. This vector defines a point in N-dimensional space that will be used by the clustering algorithm to select the most similar cluster that the point falls within. The values in the vector can be the raw data values of the selected features/sensors at a given time or can be derived values that are calculated from the collected data. As you can see in Figure 2.2, this sample vector contains voltage, temperature, and current features along with a couple of derived features that calculate the rate of change of certain sensors. IMS produces the best models when the features in each data vector are correlated in some way. For example, a data vector that contains the temperature sensors on one axis of the spacecraft will usually be very correlated and thus produce the best monitoring results.

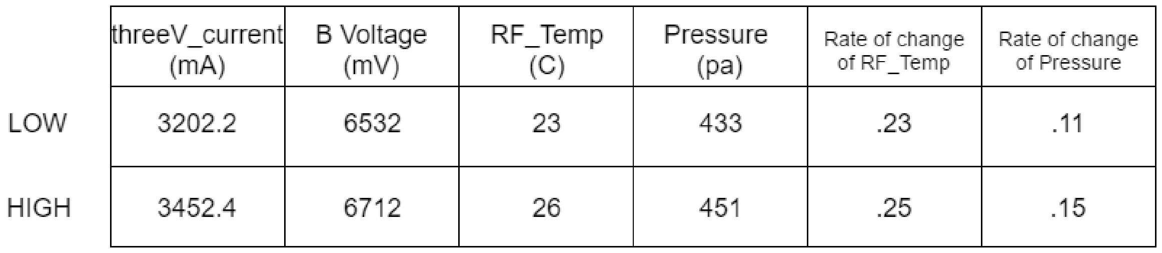


threeV_current (mA)	B Voltage (mV)	RF_Temp (C)	Pressure (pa)	Rate of change of RF_Temp	Rate of change of Pressure
3202.2	6532	23	433	.23	.11

Figure 2.2: Example of an IMS Vector

A mission controller can chose to add a large number of features in one data vector depending on how related they are to one another. The number of features does have a limit, and IMS starts to see a decline in monitoring performance as very large number of features are added due to the fact that as the number of features increase, a lot of smaller sensitive system behavior can become too generalized and lost in the model. Generally, a data vector works best when the number of features is less than 30 [13].

Now that a data vector has been defined, the next step is building the cluster knowledge base from the nominal training data. At each time point, IMS will parse the input data into the predefined vector format and use it as an input to the learning algorithm. Every cluster generated by IMS defines a range of allowable values for each feature in the input vector [11]. This cluster can be thought of as a two dimensional data vector that for each feature has a minimum and maximum allowable value. An example of this is shown in figure 2.3. The values in this two dimensional data vector define the corners of a N-dimensional minimum bounding hyper-cube where N is the number of features in the vector. Any values that fall very near or inside this bounding hyper-cube can be thought of as being nominal and part of the cluster, while ones that fall outside may be anomalous. Each cluster thus characterizes a range of values that a given nominal input vector should be near. This clustering algorithm differs from other more traditional approaches in that the clusters it generates aren't tightly defined around its points. Depending on what the low and high values are for each feature, it could produce a cluster that is too generalized and accepts new points that may be considered outliers in relation to its data points. To address this issue, a threshold value is given to the algorithm which determines how tight the bounds are in the cluster. This value is discussed in more detail next.



	threeV_current (mA)	B Voltage (mV)	RF_Temp (C)	Pressure (pa)	Rate of change of RF_Temp	Rate of change of Pressure
LOW	3202.2	6532	23	433	.23	.11
HIGH	3452.4	6712	26	451	.25	.15

Figure 2.3: Two dimensional vector that defines a cluster

Now we will take a deeper look into the IMS learning algorithm. The learning process starts with an empty cluster database. The inputs to the algorithm will be a formatted vector for a given time in the training data, and a threshold value ϵ. This threshold value ϵ defines the maximum distance that an input vector can be from an existing cluster, to be included as part of that cluster.

1. The first step is to normalize the input data vector in order to standardize the feature values and make sure they are on the same scale. Without this crucial step, some features with larger ranges will dominate over ones that have a smaller scale. Usually, standard z-score normalization is used for this step. In addition to normalization, each significant feature can be weighted so that any deviation can be amplified for that particular feature. It can be useful to do this to highly correlated features.

2. If the first vector is being processed, then it is added to the cluster database as the initial cluster. Otherwise, the distance between the vector and every existing cluster in the database is computed in order to find the cluster with the minimum distance. Several methods can be used to compute this distance, the simplest being using the Euclidean distance between the centroid of the cluster and the current vector being processed. The centroid of the cluster can simply be found by taking the average of each feature range within the vector. The algorithm will loop through each cluster and return the one with the minimum distance.

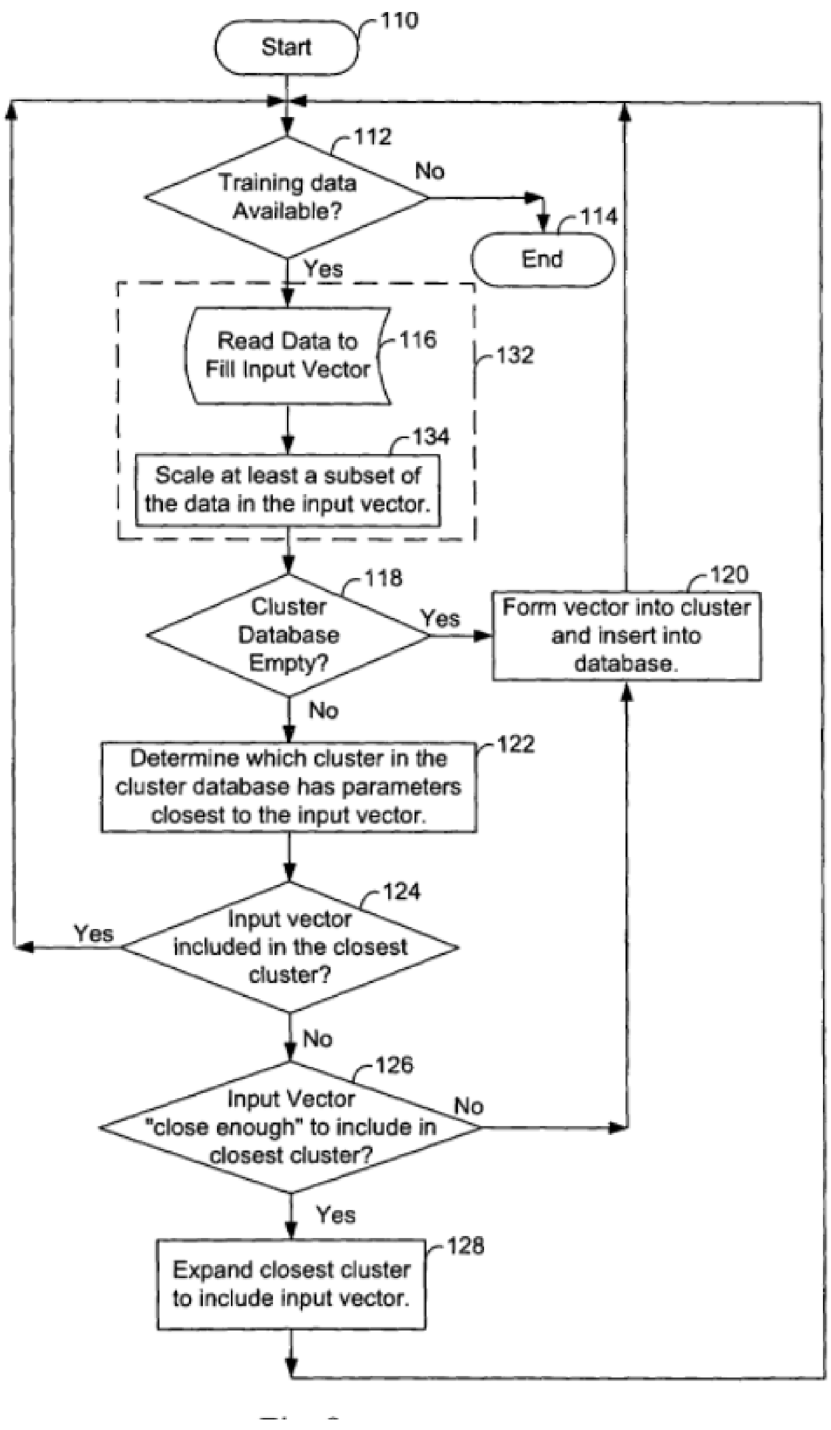
3. After finding the cluster that is closest, there will be a comparison done to see if it falls within minimum bounding hyper-cube of the cluster.

   (a) If yes, then the vector is added to the cluster.

   (b) If not, then the distance will be compared to ϵ to see if it is less than this maximum allowable radius. If it is, the vector is added to the cluster and the high and low ranges of the cluster are adjusted to include the current vector. If not, then a new cluster is created having the values of the vector as its high and low ranges.

These steps will continue until each vector in the training data is processed. The end result will be a knowledge base of all the clusters that have been generated. A flowchart of these steps is shown in Figure 2.4. This knowledge base will then be used as an input to the monitoring phase.

It is important to note that different values for the threshold value ϵ will result in a larger or smaller amount of clusters for the same input training data. The size chosen can be a trade off between tighter and better monitoring tolerances, or faster more efficient speed that may sacrifice some quality. For large values of ϵ, each cluster will have a larger radius and can thus accept more vectors. The end result will be a knowledge base with fewer clusters which are larger. This significantly speeds up the monitoring process because there are fewer clusters to process. However a higher threshold value causes the clusters it creates to be more generalized and therefore it can accept points that may not actually belong and could be considered outliers. This problem can be solved using smaller values for the threshold. Smaller values of ϵ will result in a large knowledge base of smaller clusters that are much more sensitive and can pick up more subtle system behavior. These will create clusters that better fit the points that they contain. However this smaller ϵ will cause the monitoring to be slower due to the increased number of clusters.



Start
110
112
Training data Available?
No
End
114
Yes
Read Data to Fill Input Vector
116
132
134
Scale at least a subset of the data in the input vector.
118
Cluster Database Empty?
Yes
Form vector into cluster and insert into database.
120
No
Determine which cluster in the cluster database has parameters closest to the input vector.
122
124
Input vector included in the closest cluster?
Yes
No
126
Input Vector "close enough" to include in closest cluster?
No
Yes
Expand closest cluster to include input vector.
128

Figure 2.4: Flow of how the learning algorithm works [9]

2.3.2 IMS Monitoring

Once a knowledge base has been created from the learning phase, it can be used for real time monitoring or the analysis of archived events. This monitoring produces a deviation value that signifies if the system is operating within an optimal region or may fall outside one. Large deviation values may highlight a precursor to a malfunction or a malfunction itself. This monitoring phase does not explicitly pinpoint the exact problem with the system, rather it gives details as to which features are causing the issue and where it is occurring. A mission controller can later do a closer inspection. IMS works great as an additional tool that provides an advanced method of monitoring.

IMS monitoring starts by formatting real-time data coming from the system to be monitored into the predefined data vectors from the learning phase. This data is then normalized using the same means and standard deviations that normalized the data points in the learning phase. After the normalization, the new data is in the same scale as those that were used in learning. Further scaling is done based off of the weights that were chosen for selected features in the learning phase. Once the new data vector has been normalized and weighted, the knowledge base can be queried to find the cluster that is closest. There can be two schemes for matching: strict and fuzzy [13]. In strict matching, the input vector must fall entirely within a cluster to be accepted, while in fuzzy matching a distance is calculated from the vector to each cluster and the smallest distance chosen. Strict matching is much faster as the list of clusters needs to be only traversed once to see if the given vector falls within. One drawback of strict matching is that if the training data isn't comprehensive, than the system may give an otherwise nominal vector an anomalous result. Fuzzy matching on the other hand, is much slower since a distance needs to be calculated between the input vector and each cluster. Distances that are close enough to a given vector may indicate behavior that is nominal but not contained within the training data. Overall the fuzzy scheme is much more robust, and used much more often.

The full process to obtain this deviation value for a real-time system works as follows:

1. Perform real-time data acquisition for the parameters that were selected for monitoring in the learning phase and place them in a new data vector that will be used for monitoring. It is important that the format of the data vector is exactly the same as the one used for learning; otherwise, the algorithm will try to relate different parameters. A threshold value ϵ is also given as input, which specifies an acceptable limit that the deviation score can be to be accepted as nominal.

2. Normalize the newly created data vector with the same scale that was used in the learning phase. This may involve saving the means and standard deviations for each feature in the training data and then using these values to perform the normalization on new values as is the case in z-score normalization. In addition to normalization, if feature weighting was used in the training phase, then the same weights must be applied to the new monitoring vector.

3. Once the data has been normalized and scaled, the knowledge base will be queried to find the cluster that is closest to the data vector. The algorithm will loop through each cluster in the knowledge base and calculate the distance between the monitored vector and the closest edge of the clusters minimum bounding hyper-cube. This distance can be found by simply using the Euclidean formula for distance for two n-dimensional points shown here where x and y represent the two points.

   $$d = \sqrt{\sum_{i=0}^{n} (x_i - y_i)^2}$$

   If the vector falls within the nominal operating region defined by the n-dimensional hyper-cube, then the distance returned is zero. If one or more parameters within the vector do not fall within the cluster, then a non-zero value will be returned indicating the deviation value.

   During this step, the individual sums for each feature that contribute to the overall distance can be saved. This allows the operator to get a more detailed look at which sensors are problematic. Parameters that are contributing higher sums are farther from the hyper-cube in that dimension, which indicate that the particular sensor isn't conforming with the model.

4. The cluster with the lowest distance is then checked to see if that data vector falls inside its nominal operating range. If it is, then the the system is performing nominally. If not, then this distance is compared against the threshold value. If the value is greater, then an alert can be sent that warns mission controllers that there may be a problem with the system. There can potentially be multiple alert levels based off this deviation value with higher values indicating major malfunctions.

Mission controllers can have special tools, such as Graphical User Interface's, that can graph the overall deviation scores over time and also allow for easier trend analysis of the individual parameters. It is important to note that while IMS does a great job at finding potential issues with the system, it doesn't fix them. Once the problem has been identified, a course of action needs to be made by mission controllers to rectify the issue.

Overall, IMS's monitoring capabilities are robust and powerful. It provides a means to capture nominal system behavior by creating models that correlate the behavior of multiple sensors. The sensitivity of these models can be adjusted based on the training data used and the threshold values given so that there can be a balance between speed and accuracy. IMS is also very adaptable for system monitoring applications. The knowledge base that the monitoring algorithm uses can be updated at any time to provide a more accurate model of the system as more nominal telemetry is gathered. The features that are monitored can also be updated to remove or add new features that may provide better results. The strengths of IMS have led it to become very successful in a number of system applications.

IMS will be the machine learning algorithm used in this thesis to create a real-time health monitoring system for CubeSats.

2.3.3 IMS Applications

The potential applications of the Inductive Monitoring System are not limited to the monitoring of spacecraft. Any domain which has access to large archives of previous operational data could use IMS as a health monitoring tool. IMS has seen significant success in its application across these fields as a tool that can actively find anomalies in data that were previously undiscovered. IMS was originally created for spacecraft applications, and was initially part of NASA's Integrated Vehicle Health Management suite that ran on board the vehicle or in the mission control room to provide monitoring of the vehicle.

STS-107 Analysis

IMS's utility was first proven useful by its analysis of the Space Shuttle Columbia (STS-107) disaster that occurred on February 1, 2003. The space shuttle orbiter was destroyed during re-entry into the atmosphere and killed all seven astronauts on board [8]. An investigation into the incident concluded that a large piece of foam was released from the shuttle's external tank 82 seconds after takeoff and hit the leading edge of the left wing which caused a breach in the Thermal Protection System of the spacecraft. No damage to the orbiter was noticed by mission control until 17 days after launch, and was a small increase in the brake line temperature of the left landing gear that was seen only seven minutes before the spacecraft was destroyed [11]. Existing monitoring tools on the spacecraft failed to detect this anomaly until it was too late.

Because of this disaster, it was clear that more advanced tools were necessary to enhance the monitoring capabilities of the spacecraft. To demonstrate that the newly developed IMS tool could provide advanced monitoring of such systems, Iverson used archived telemetry from the temperature sensors of the wings of previous successful space shuttle missions. Through this training data, multiple knowledge bases of nominal operating regions were created from the temperature of the wings during several phases of the flight including launch, ascent, and re-entry. These knowledge bases were then utilized to analyze the archived data of the STS-107 flight and see if it was able to pick up any anomalies [11].

The training vectors were formed from four temperature sensors on each wing of the shuttle. These vectors were normalized and then used for the generation of a separate knowledge base for each wing. The end results was a knowledge base of 490 clusters for the left wing and 237 clusters for the right wing.

After feeding the flight data of STS-107 into IMS, the output in Figure 2.5 was produced. The horizontal axis represents the time from the beginning of lift off, while the vertical axis represents the deviation value from nominal that the analysis produced for a certain time. The blue line represents the deviation values of the right wing while the pink line represents the deviation values of the left wing. The moment of the foam impact is also marked by a small vertical line at about 15:40:22. Looking at these results, it can be seen that before the impact, the deviation values for both wings were trailing each other and were quite small. However after the impact the left wing saw huge spikes in the deviation value that indicated some type of error or malfunction. Closer analysis of the individual temperature sensors that were causing these spikes would have alerted mission controllers to this problem much sooner. These results show that IMS can be effectively used to provide advanced monitoring and detect system anomalies. This analysis was done post-disaster on the archived telemetry of the mission. However it can also be used for real time monitoring of the spacecraft to provide mission controllers with more monitoring capabilities [11].

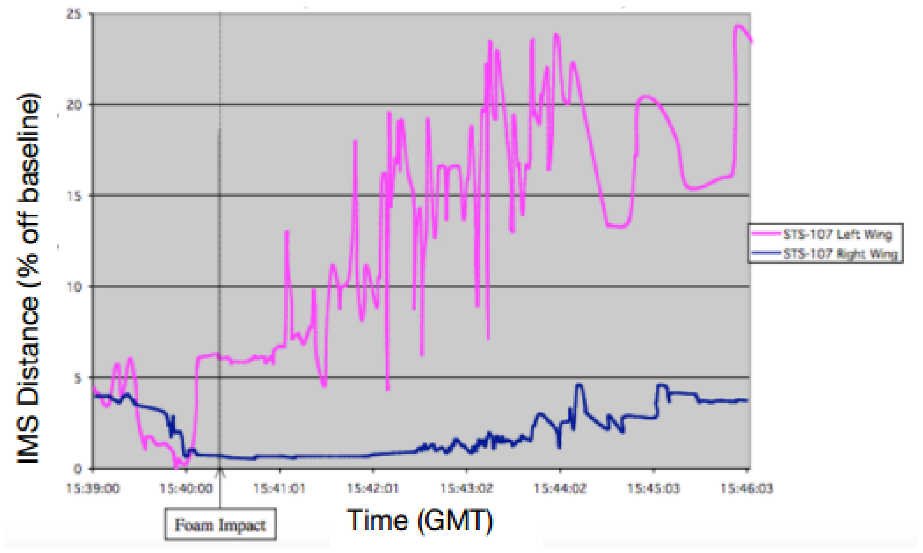


Figure 2.5: Results of IMS analysis of STS-107 [11]

International Space Station Mission Control

IMS has been further matured and slightly adapted for use in the real time monitoring of several application including the International Space Station(ISS) flight control room. For this application, IMS provides monitoring of two flight control disciplines, Attitude Control and Thermal Operations [10]. Four large gyroscopes make up the ISS Control Moment Gyroscope (CMG) attitude control system which provide the ISS with "non-propulsive attitude control devices that exchange momentum with the ISS through induced gyroscopic torques. [10]" These systems have been known to degrade throughout time enough to malfunction, and thus need to be replaced. Due to this, flight control officers are interested in discovering any early symptoms of degradation so that replacements can be scheduled.

IMS was deployed in the assistance of this task to provide real time monitoring of the CMGs degradation. The IMS data vectors for this application consisted of nine sensors and four derived parameters that were chosen with the help of special flight controllers. Some of these parameters include rotation speed, bearing temperature, rotation rates, and rate of change of the temperature and current sensors over time. In order to generate the models for the nominal operating regions, seven to ten months of archived telemetry was used from four CMGs to capture their unique characteristics. The result was four IMS knowledge bases that were generated from the data that would be used to monitor each CMG.

This IMS system was integrated with the NASA Mission Control data server software so that it could access real-time telemetry coming from the ISS. Each CMG had its own IMS module running to provide continuous real-time monitoring of the incoming telemetry to the appropriate CMG knowledge base. Each of these IMS modules would report the overall deviation of the telemetry from the nominal regions and could also provide the individual deviation contribution of each parameter so more concise analysis could be done and the source determined. These results are posted on the ISS Mission Control data server which can then be plotted on console displays and alerts issued if significant deviations occur. The success of this system has led to further advancement and development of the system to provide more specialized monitoring capabilities. Another significant advantage of the system is that it can be updated frequently as new telemetry data sets are archived and provide more details of the overall system.

2.4 CubeSat

CubeSats are small pico-satellites that are designed to reduce costs and development times, increase access to space, and be able to sustain frequent launches with launch providers [14]. The CubeSat project began in 1999 as a collaborative effort between California Polytechic University, San Luis Obispo and Stanford University's

Space Development Laboratory [19]. They were created so that educational institutions and small private firms could have the capability to create small affordable satellites that could be developed in 1-2 years and each perform a specific mission. The CubeSat standard defines the shape and size of one unit(U) to be a 10 cm cube with a mass of up to 1.33 kg per unit [14]. A CubeSat can be multiple units long as is the case with 2U and 3U CubeSats as long as each unit follows the standard. Creating larger CubeSats can allow for missions of increasing complexity and value, however can lead to increased costs and development times. The power and size constraints of CubeSats can introduce challenges in terms of hardware/software design and certain trade-offs have to be made to balance the cost, effectiveness, and fault tolerance of these systems [16]. Most CubeSats consist of a unique payload that performs the science experiments that are required for the mission. CubeSats are deployed using the Picosatellite Orbital Deployer (P-POD) that was developed at Cal Poly which protects the satellite from the launch vehicle and provides deployment capabilities when in orbit [14].

Each CubeSat can have a plethora of hardware components and micro controllers that as a whole constitute the spacecraft. Most components have some type of sensor that can give the ground station more information about its state. This information can include the temperature, current, and voltage of certain components. These sensors can be analyzed to detect and locate any malfunctions and monitor the overall health of the satellite. Overall, due to their low costs and development times, CubeSats have seen widespread development from different universities and private firms including PolySat which has been a leader in the development of CubeSats since their inception.

2.5 PolySat

PolySat is a multidisciplinary lab run by students on Cal Poly's campus devoted to the design, implementation, testing, and integration of CubeSats. The lab has successfully launched 8 CubeSats since the start of the program and each mission has given new insights on how to improve both the hardware and software design of the spacecraft [3]. The current design of the lab's avionics system relies on a a few key design principles that make the spacecraft economical, radiation resistant, easy to develop for, and match the size and power constraints.

The hardware is designed to minimize modularity and redundancy in order to make it more generalized for more missions and to reduce the space the base hardware occupies [16]. This design has led to the consolidation of most of the spacecraft's hardware into a single system board which contains both the command and data handling and electronic power supply subsystems. This system board consists of a small but powerful 32-bit ARM9-based Atmel processor which runs at a clock rate of 400MHz. It also contains 64 MB of S-RAM, 512MB of non-volatile NAND flash memory, and 16 MB of non-volatile phase change memory which is far more radiation tolerant than other forms of memory. Radiation tolerance is very important for CubeSats because the environment in space is susceptible to radiation events which can damage hardware and corrupt the software. For additional storage, the spacecraft contains a SD card with a capacity of 32 GB [15]. Many standard serial communication protocols such as I2C and SPI are used to connect and communicate with the different components and sensors that make up the spacecraft.

There is also a certain level of fault tolerance built into the CubeSats developed by the lab. This capability allows for the spacecraft to recover from any transient faults that occur during the mission and impact the operations of the spacecraft. To provide this fault tolerance, there are a number of hardware and software watchdog's that periodically check if the spacecraft has reached some kind of error state. If so, the system is rebooted in order to recover from the error state and continue operation as normal. Checksum's are also used to verify that persistently stored data hasn't been corrupted [16]. While these fault tolerance systems provide the spacecraft with a good level of protection from transient faults that can occur in the harsh space environment, they fail to identify and detect any major hardware malfunctions that may have occurred.

PolySat currently has no health monitoring system that exists within the flight software that allows for the monitoring of the spacecraft while in orbit. The current method of monitoring involves collecting large chunks of system telemetry that are stored on an onboard SQLite database and analyzing each sensor to try to find the cause of a problem. Since the downlink capability of the spacecraft is severely limited, transferring these large files can be time consuming. It can also take a long time to analyze the data for potential problems. Some of the current PolySat missions have over 200 sensors on board that would have to be examined. A subtle problem would be extremely hard to detect by just eye balling the data. An automated health monitoring system could greatly enhance the way the lab detects malfunctions during the satellite's mission time frame. To do this, IMS will be implemented within the flight software to provide this capability that is much needed. In this section, we will be taking a look at the flight software design of PolySat to better understand how IMS will be integrated.

2.5.1 Software Design and Architecture

PolySat's flight software was designed to be highly modular, extensible, and robust so that it can be used reliably for many missions. These design goals allow for reuse of large portions of the flight software for new missions instead of tailoring the software for a specific mission. Modularity improves the overall organization of the software by isolating major spacecraft functions into their own modules. This modular design has a number of important benefits including speed, simpler debugging and maintenance, ability for parallel development, and extensibility. It is much easier to debug problems that may be occurring in a modular architecture because the problem will be isolated within one or two specific processes without affecting the whole system [15]. Another big advantage is that it allows for parallel development by multiple students where each student can work on different modules at the same time. Since CubeSat missions have a short turnaround time, development speed is an important factor to consider and parallel development reduces this time by a substantial amount. Also since each major function of the satellite is its own module, full knowledge of the system is not necessary and a student developer can begin contributing much sooner.

To support some of these goals and make development easier, the flight software is built on top of the Linux operating system. Linux provides a large amount of pre-existing code and libraries that handle the low-level tasks of managing the hardware, drivers, and communication [16]. Linux's process model also supports the goal of modularity by providing address space and code isolation of major spacecraft tasks into separate processes. As a result, PolySat's flight software consists of several core processes that handle the major tasks of the system. The overall software architecture consists of three main layers: processes, abstraction libraries, and drivers. Since most processes perform similar tasks such as event scheduling and command handling, abstraction libraries provide a mechanism to provide these common services to each process. The flight software also operates in an event-driven fashion where processes block until there is some timed or command initiated event that causes something to happen. A high level overview of the different components that make up the PolySat flight software can be seen in Figure 2.6.

2.5.2 Processes

The flight software is broken down into a number of core processes that perform the major operational functions of the system. Each process performs a very specific function and uses the custom PolySat abstraction libraries to perform common tasks. Communication between processes is done using Inter-Process communication which leverages the UDP/IP Networking Protocol built into the Linux kernel [15]. Each process in the flight software is assigned its own port and any other process that wishes to communicate simply generates a UDP packet with a destination at the given port. UDP and IP also contain built in error detection by utilizing checksums which allow for a much more reliable method of transmission.

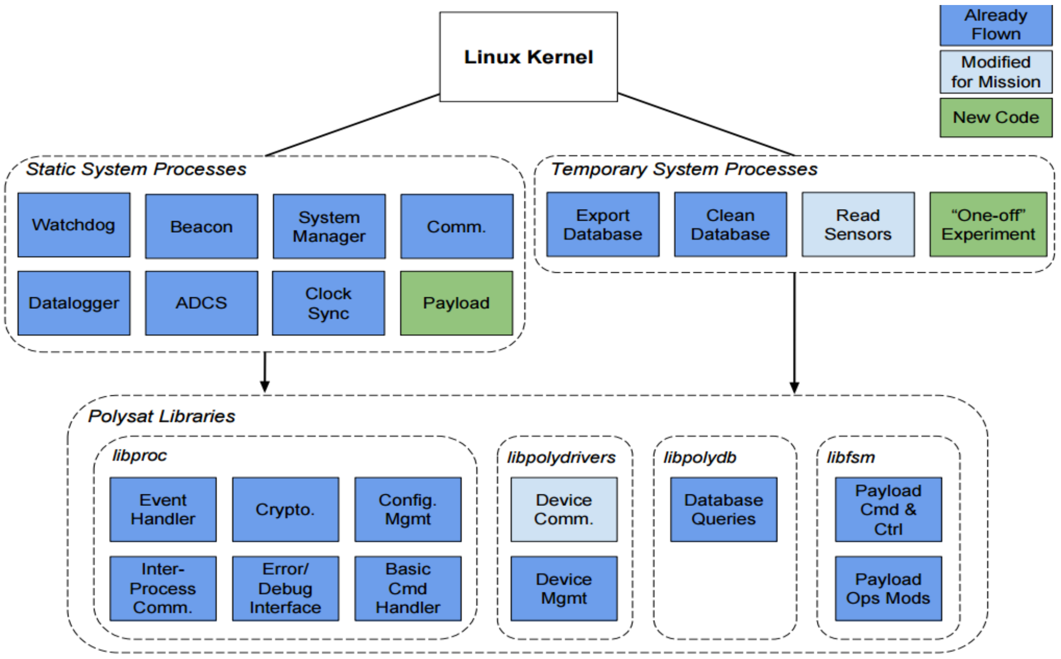


Figure 2.6: High-level overview of the different components in PolySat's Flight Software

Some of the main processes include:

1. **System Manager -** Responsible for maintaining the state of the avionics system, which includes various kernel statistics, hardware state information, and

sensor information. It also has the capability to give administrative actions that can do important tasks such as killing a specific process or rebooting the satellite. It is also responsible for the reading and storing of spacecraft telemetry.

2. **SatComm -** Responsible for controlling the radio and sending/receiving data from the ground. SatComm handles the encoding and decoding of the packets from the transceiver and provides the low level network management required to send and receive data from the ground station. SatComm also is responsible for the routing of packets within the spacecraft itself and uses Linux's built in networking libraries to allow for the Inter-process communication that is essential for the spacecraft to communicate with itself.

3. **Beacon -** Periodically broadcasts spacecraft identification and health information. The Beacon process is very important and allows for the tracking and identification of individual CubeSats in orbit which can be a difficult task if multiple spacecraft are released at the same time. The packet broadcast by the Beacon process contains essential information about the health of the system and can be received and decoded by the ground station or via any amateur radio station.

4. **Watchdog -** Provides a mechanism for the software fault tolerance of the system. As mentioned before, there are many radiation events that can occur in the space environment that can cause bit flips and data corruption. These events could cause a process to behave abnormally or reach an error state. The software watchdog periodically queries each process and validates if they are working correctly. If not, it causes a system reboot which will hopefully solve the issue.

5. **Datalogger -** Responsible for administration and storing of all telemetry in the SQLite Database. The storage of telemetry in the the database is an important

event that occurs at a scheduled interval throughout the mission and Datalogger handles this task. It can also store other data that may be generated from the scientific mission in the form of key value pairs. All this data can be later accessed and down-linked for further analyzing.

There are also other processes that may be mission specific such as a payload process that interfaces with the mission payload. There are also other temporary processes that can manipulate the database or perform "One-off" experiments. Each process also has command handlers which handle any commands that may be transmitted by the ground station or another process. Each command may perform a specific function that may be specific to a process.

2.5.3 Abstraction libraries

There is a large set of functionality that is common across all processes. Examples of this include event and command handling, inter-process communication, and configuration management. This common functionality is provided by a standard set of custom libraries that expose an API that processes can use. By using these libraries, the development of the process is significantly easier and faster.

The main libraries consist of the following:

1. **Event handling-** As mentioned earlier, the spacecraft is a event-driven system that responds to various commanded or timed events. Each process must therefore have an event handler that can generate events or respond to events that may occur. The event handling system is provided as a library that each process can use to provide this functionality. The library takes advantage of the Linux select call which can monitor various file descriptors in the system and also wait for timed events. Since each process is assigned a socket file descriptor,

this file descriptor can be used by the select call to register commands. Each process will block until either a file descriptor is set or a timed event occurs. The library provides a method to register a callback function to be called once either of the two events occurs so that process can respond accordingly to the event. This library abstracts all of the details into a concise set of API calls that sets up the event handler for the process and can schedule events.

2. **Command handling-** Any spacecraft must have the capability to respond to commands that are sent from the ground or any subsystem on board. Command handling is done on each process by using a library that maps command numbers to functions. Commands are sent to a pre-designated port number assigned to each process during initialization, and consist of a one byte command number that is appended to the beginning of a command packet. This command number is used as an index into an array of function pointers that provide the functionality required of the process for that command [15]. This mapping is defined using a command configuration file that provides details on each command that a process can handle. An example of a common command that is seen across most processes is a status command that returns state telemetry of the selected process. Each command returns a unique command response number as part of its packet to identify which command handler is responding.

3. **Inter-Process Communication -** Inter-process communication between processes within the satellite is also abstracted into a library that is utilized by every process. This library utilizes existing Linux API calls to create non-blocking UDP sockets that are used by the command handler to send and receive commands. It also provides an abstraction to look up the port number of any process using the process name.

PolySat's flight software adheres to the main design goals of modularity, extensibility, and robustness that make development much more streamlined and organized. The success of this system has been seen in its utilization in multiple missions.

Chapter 3

RELATED WORK

The algorithms in the Inductive Health Monitoring System are not the only ones that can be used for anomaly detection and monitoring in this application. There are several other unsupervised and supervised algorithms that can be used for outlier and anomaly detection. Many of these algorithm are also data-driven and require archived training data to generate some nominal representation of system behavior. Some examples of such algorithms include ORCA, One-class SVM's, and Virtual Sensors. In this section, some of these alternative algorithm are examined and compared to IMS.

3.1 ORCA

ORCA is a tool that also uses a data driven unsupervised algorithm for anomaly and outlier detection [6]. It is distance based like IMS and uses a nearest neighbor approach to calculate the average distance between a point and its k nearest neighbor's that reside in the feature space around it. This distance is reported as the anomaly score that can be used to determine if the new point is an anomaly or not. It is a very fast and efficient algorithm that uses a simple pruning rule that creates near linear time performance on large data sets.

One of the big advantages that ORCA has over IMS is that it can be used to find outliers in a large heterogeneous data set [10]. This can be useful when trying to find outliers in the training data. IMS relies on all data points in the training data to be nominal. If there are any anomalous points, then this bad behavior is included in model that IMS generates. ORCA can identify these outliers within a data set and therefore doesn't suffer from this problem of including bad data in its models. ORCA can be used in conjunction with IMS to remove all outliers from a training data set before running the IMS algorithms.

3.2 One-class Support Vector Machines

One-class Support Vector Machines (SVMs) use training data with one class, in this case nominal data, to create a model which is used for anomaly detection [21]. They can be used to separate anomalous data from nominal data. They are a subset of a Generic Support Vector machine, which is used in classification to classify data by finding a decision boundary which separates each class from the other classes. It finds these decision boundaries by mapping data from a lower dimension to a higher dimension so a linearly separable hyper-plane can be found that splits each feature. In the case of one-class SVMs, this hyper-plane separates points that are deemed nominal from those that are anomalous. An example of this separation in high dimensional space can be seen in Figure 3.1, where the purple dots are nominal observations whereas the yellow dots represent an outlier. These SVMs return a measure of strongly a new data point is nominal or anomalous.

Since these SVMs are not distance based in the same sense as ORCA or IMS, the points that it finds anomalous often differ from these distance based models. One-class SVMs calculate the distance from a point to its separating hyper-plane instead of analyzing the data and finding distances in the original data space [17]. IMS and ORCA are more robust and better at monitoring applications that may have significant mode changes throughout operations because their models use the characteristics of the original data space to define anomalousness.

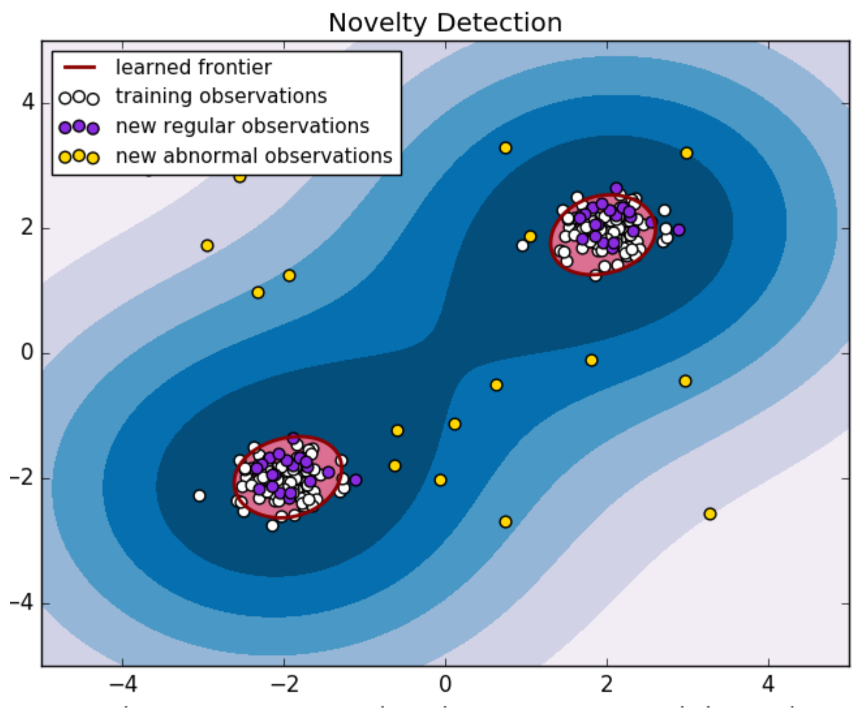


Figure 3.1: One-class SVM [2]

3.3 Virtual Sensors

Virtual Sensors is a supervised machine learning algorithm that can be used for anomaly detection and differs from IMS's one class monitoring scheme. This algorithm predicts the values of certain features at time t using available data up to and including the given time. It does this prediction by using a non-parametric model to create an approximating function that predicts the value of a specific output variable as a function of other observed variables and an input vector. This predicted value is compared against an observed value to determine an error value which defines how anomalous the given input vector is [18].

Since this algorithm uses adaptive modeling to create an approximating function, it needs a way to adapt to changing system operating modes that cause the data to change significantly. To address this issue, multiple models are created for different operating modes and a distribution of the estimates is used to generate the error value that takes into account any transient mode of operation changes. Both IMS and Virtual Sensors produce similar results when used to detect anomalies. Differences in the ways these algorithm determine errors may causes either algorithm to find an anomaly that the other may not detect.

Most of the algorithms that have been discussed in this section can be used alongside or replace IMS as a potential anomaly detection algorithm. The best results are obtained when multiple algorithms can be used alongside each other so any anomaly that is hard to detect in one model, can be captured in another.

Chapter 4

SYSTEM OVERVIEW

PolySat's system health monitoring abilities are only limited to detection of problems that are glaringly obvious through its beacon or manual examination of flight telemetry. These methods are not only inefficient and time-consuming for mission controllers, but also require increased bandwidth to send large amounts of telemetry which could be utilized more effectively for mission tasks. Also, as the complexity of future CubeSat missions increase, it will become increasingly difficult to find small problems that effect mission operations and results.

Fortunately, PolySat has collected a large archive of system flight data throughout many missions that make it possible to use data-driven monitoring techniques such as IMS to monitor the health of the satellite. By using this technique, we gain all of the advantages IMS offers such as simplicity, adaptability, and efficiency. IMS is used in this thesis to create a flight ready and validated implementation that will be used in all future missions. This section discusses the requirements and high-level design of the IMS system that is integrated into PolySat's flight software.

4.1 Requirements and Goals

For this new health monitoring system to be successfully integrated in future missions, it must meet a set of requirements. These requirements ensure that the system doesn't negatively impact spacecraft operations, and can provide the best monitoring results throughout the mission cycle. These main requirements can be listed as follows:

- **Speed and efficiency -** The health monitoring system should be able to process and output the deviation score in real-time. Therefore the monitoring algorithm should be fast and efficient during operation.

- **Low resource consumption -** The system should not require a significant amount of resources in terms of memory or processing power from the satellite. CubeSat's are very power limited due to their size and therefore power must be efficiently used. A lot of processing from the system increases this power draw and may cause power issues. The memory of the CubeSat is also limited to 64MB to be shared among all processes. The health monitoring system should use the least amount of memory possible in order to conserve this limited resource.

- **Anomaly reporting capability -** The system must be able to provide alerts when the spacecraft may not be functioning nominally. Along with these alerts, it should provide more details about the severity of the issue and possible causes.

- **Generic -** The system must be generic so that it can be used in multiple missions with very little changes. Since each mission is different in terms of design and complexity, the system should be able to adapt to the new hardware and sensors each mission introduces. It can be used by any process on the flight software to provide specific monitoring.

- **Adaptability -** The system must be able to be updated as the mission progresses. These updates can improve upon the existing model as new nominal flight telemetry is gathered. Since it is highly unlikely that complete system behavior is captured for a specific mission before flight, updates on orbit can fill in the missing holes to provide better results. The system must also be able to be shut down in case it is performing poorly.

The overall goal of this health monitoring system is to provide an autonomous monitoring tool that PolySat mission controllers can use to check if the spacecraft is operating nominally. This system could potentially catch issues far before they become apparent and cause real damage to the spacecraft. It will also ensure that any scientific results that are being down-linked haven't been influenced by some malfunction.

4.2 High level design

The new health monitoring system needs to be designed in a way so that these critical requirements can be met. Fortunately, IMS is a great system to use which meets most of these requirements. This new system was designed around IMS's learning and monitoring phases, and uses the same algorithms that were discussed in Section 2.3. The success and speed of these algorithms on some of IMS's previous applications made us confident to use it for a spacecraft wide monitoring application.

The high level design of this system consists of a learning phase which occurs on the ground, and a monitoring phase which occurs on the spacecraft during its mission. The idea behind this separation is to perform the more resource intensive learning phase on a computer on the ground, and then load the knowledge base onto the spacecraft so that it can perform the less intensive monitoring. The monitoring returns a report on the health status of the system. This design works well to meet the speed and resource requirement listed. The steps can be summarized as such:

1. Gather all nominal telemetry from the archives
2. Select features to monitor
3. Run the learning algorithm on selected features and archived data
4. Upload the resulting knowledge base of clusters onto satellite

5. Run monitoring

6. Report health status

The model that is generated during the learning phase can be updated as the mission progresses and the spacecraft down-links relevant nominal telemetry. This new telemetry is added to the archive, and a new model is generated that represents a more comprehensive and accurate representation of system behavior. This cycle can continue throughout the mission. This achieves the requirement of adaptability. This high level design is illustrated in Figure 4.1.

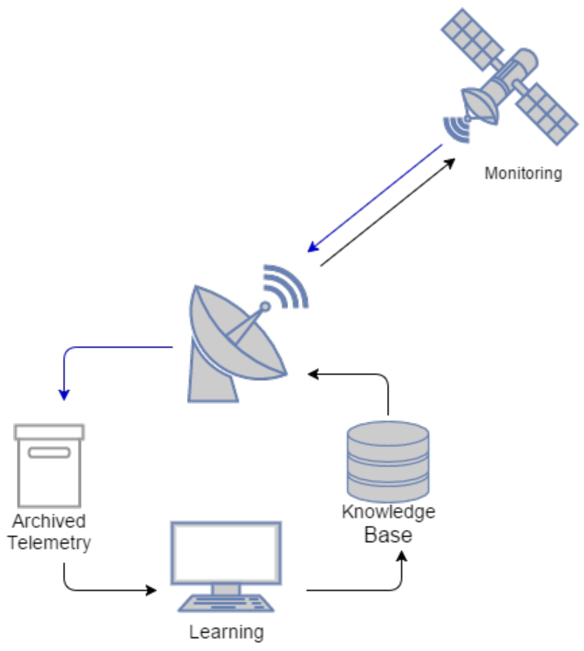


Figure 4.1: High level system design

Reporting of the health status will be done through the satellites beacon packet which broadcasts periodically as discussed in Section 2.5.2. This beacon packet can be received by the ground station and the health report read. Additionally, the monitoring algorithm's deviation score and individual feature contributions can be saved on a file on board that can be down-linked for further analysis.

The final system will consist of multiple IMS modules that define different feature sets to be monitored. Each module will be trained separately on the ground and run independently on the satellite. Each module will belong to a process on the satellite and the monitoring capabilities will be provided as a library that the process uses. During monitoring, feature values will be obtained by sending UDP commands to the processes that own that particular feature. Once all feature values in a module are available, the monitoring algorithm will process the data and return a deviation score. More details on this system will be discussed in the next sections.

4.3 Config/Cluster files

A standard structure and format of the knowledge base is important so that the output of the learning phase can be easily read by the monitoring library. In Section 2.3.1 we see that the learning phase will generate a knowledge base containing all the clusters that define the nominal behavior of each feature set. To provide the best monitoring results, multiple feature sets or ”monitoring modules” will be created and trained to run as separate models on the satellite.

These modules need to be explicitly defined so both the learning and monitoring phases can run on the correct features. This is achieved using configuration files that are built into PolySat’s software libraries. These configuration files will be generated by lab members and each file will define important information about the specific module including its name, cluster file location, the process is belongs to, and details on all of its features. It will be used to select which features to train on from the archived telemetry during learning. The real power of these files comes from its use by the monitoring library on the satellite to define the proper modules and initialize the proper data structures required for monitoring. These configuration files have a style similar to XML and consist of tags and values.

In addition to the configuration file, the actual knowledge base or cluster files generated by the learning phase are required for monitoring. These also need to be saved and correspond to the correct configuration file which defines its feature set. The cluster file will be a large data file consisting of floating point numbers that represent the clusters. The configuration files define the modules while the cluster files provides the details and data of the module. This generates a file hierarchy that the spacecraft will use to perform the monitoring. This is illustrated in Figure 4.2.

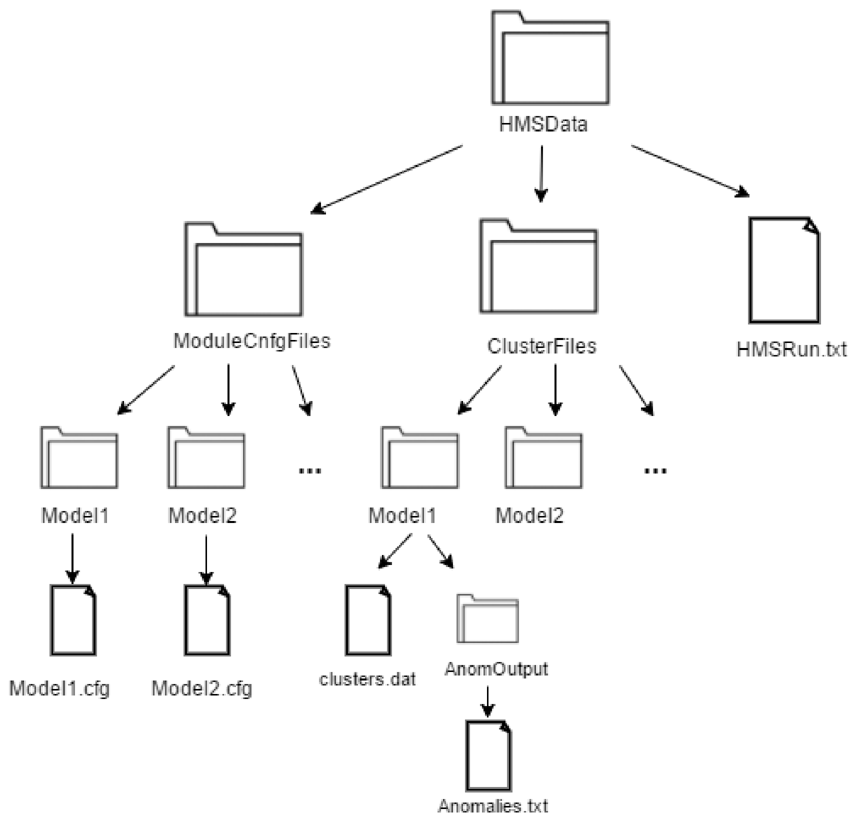


Figure 4.2: Cluster and Configuration file hierarchy

This hierarchy begins with a root directory *HMSData* which consists of subdirectories that contain the configuration and cluster files, *ModuleCnfgFiles* and *ClusterFiles* respectively. This directory also contains a *HMSRun.txt* that specifies whether or not the monitoring should be running. The configuration and cluster file directories contain multiple sub-directories for each module defined. For example, if three modules were created, there would be three subdirectories with the names of the module under both the *ModuleCnfgFiles* and *ClusterFiles* directories. These directories then finally contain the respective configuration or cluster file for the module selected. Finally, each *ClusterFiles* module sub-directory contains a *AnomOutput* sub-directory which stores detailed information on any anomalies that may have occurred for that specific module. One of the reasons for this seemingly excessive and complex file hierarchy has to do with the adaptability of the system which will be discussed more thoroughly later. This file structure will be copied to the */data* directory on the file system of the satellite, and be used to initialize the monitoring structures and perform the anomaly detection.

4.4 Learning Module

One of the first steps in this health monitoring application is to create the models that will represent nominal system behavior for the satellite. This is done using IMS learning and will be performed on a local computer on the ground. The output of this learning phase is the hierarchical file structure mentioned in the previous section.

IMS learning requires large amounts of archived nominal telemetry to train its models and generate cluster files that characterizes most of the system behavior. PolySat has gathered large amounts of flight telemetry from its past missions that can be used for this purpose. It is important that this data contains no anomalies so that bad behavior isn't included in the final model. As a result, portions of this data have been analyzed and filtered of any potential anomalies. In particular flight data from the Intelligent Payload Experiment (IPEX) mission was used to generate the models necessary. This data set consists of about 50,000 time series data points taken about every 10 minutes that each contain data for over 150 features. This large number of features includes hardware sensors that monitor temperature, current, voltage, pressure, etc, and a number of software telemetry points that give various statistics about the state of the software.

4.4.1 Feature Selection

Given this large number of features, it is best to not just pack them all into one model. IMS works best when features are correlated and changes in one feature in the feature set have a predictable change in another. Also one big benefit of having a feature set that is highly correlated is that a smaller amount of training data is needed to capture the behavior for that set. It is also not good practice to have more than 25-30 features in one model due to it becoming too generalized and losing specific system behavior. For these reasons, it is best to break up this data into multiple correlated feature sets that can be used for monitoring. In order to do this, a technique to find the correlation of each feature in this data set will need to be used. For this application, we take advantage of the power of a correlation matrix.

Covariance is a statistical measure of the linear association between two random variables x and y. It measures the strength of the correlation between these two variables and is defined by the formula:

$$f(x) = \sum_{i=0}^{n} \frac{(x_i - \overline{x})(y_i - \overline{y})}{n}$$

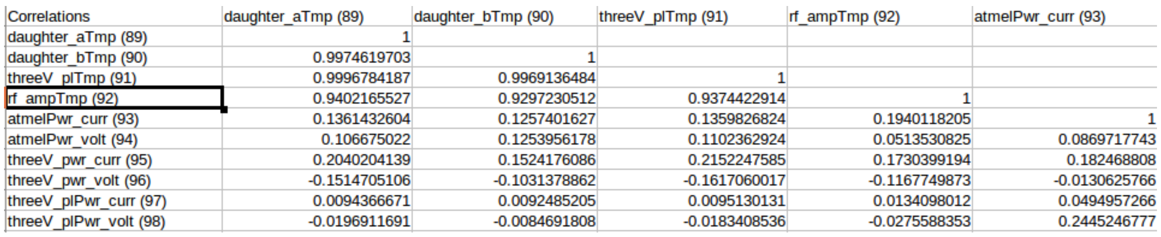
This formula basically finds the sum of the pairwise data points that fall on the same side of the mean or on opposite ends. The covariance of two variables is positive when this sum indicates that most data points fall on the same side of the mean, and negative otherwise. The strength of the correlation, either positive or negative, is determined by the absolute value returned. The larger the value, the stronger the linear relationship between the two variables [23].

Covariance is highly influenced by the scale of the data points and this influence needs to be removed. To find a scale-free metric of this linear relationship, a correlation coefficient is used. The correlation coefficient r is a normalized measure of the linear relationship between two variables and is defined as the covariance of x and y divided by the standard deviation of x multiplied by the standard deviation of y.

$$r = \frac{cov(x, y)}{\sigma_x \sigma_y}$$

The correlation coefficient is a much better metric to use to find the correlation of two or more variables in a large dataset because it is scale free. A correlation matrix is created by finding the pairwise correlation coefficients between each variable in the dataset. Highly correlated pairs of variables will have a correlation coefficient close to 1 or -1. Pairs with low correlation will have values closer to zero.

For our application, we want to find instances of either strong positive or negative correlation between variables and group them into the same module. A correlation matrix was generated for the hardware sensors in the IPEX training data and the correlation coefficient between each pair of features was calculated.



Correlations	daughter_aTmp (89)	daughter_bTmp (90)	threeV_plTmp (91)	rf_ampTmp (92)	atmelPwr_curr (93)
daughter_aTmp (89)	1				
daughter_bTmp (90)	0.9974619703	1			
threeV_plTmp (91)	0.9996784187	0.9969136484	1		
rf_ampTmp (92)	0.9402165527	0.9297230512	0.9374422914	1	
atmelPwr_curr (93)	0.1361432604	0.1257401627	0.1359826824	0.1940118205	1
atmelPwr_volt (94)	0.106675022	0.1253956178	0.1102362924	0.0513530825	0.0869717743
threeV_pwr_curr (95)	0.2040204139	0.1524176086	0.2152247585	0.1730399194	0.182468808
threeV_pwr_volt (96)	-0.1514705106	-0.1031378862	-0.1617060017	-0.1167749873	-0.0130625766
threeV_plPwr_curr (97)	0.0094366671	0.0092485205	0.0095130131	0.0134098012	0.0494957266
threeV_plPwr_volt (98)	-0.0196911691	-0.0084691808	-0.0183408536	-0.0275588353	0.2445246777

Figure 4.3: Correlation matrix for small set of features

Figure 4.3 illustrates a small set of the entire correlation matrix. We can see that there seems to be strong correlations between each of the temperature sensors and low correlation between a temperature sensor and power sensor. This relationship was seen throughout the rest of the matrix. This relationship makes sense because we expect sensors in the same vicinity to experience the same temperature changes. Strong correlations were also seen between most of the training data's power sensors. These results led to the creation of two main modules for monitoring: one that contains a large set of temperature sensors, and the second which contains most of the power sensors. The configuration files that define these modules can be seen in Appendix A.

4.4.2 Output

Once these feature sets were determined, the learning algorithm could be run on each module to generate the cluster files. The IPEX data would be parsed to include only the selected features in the module, and then the clustering would occur on this parsed data. The resulting cluster file includes the clusters, along with some scaling information needed for the data normalization such as the means, standard deviations, and weights of each parameter. More information on this will be provided in the implementation section.

Along with the cluster file, a configuration file needs to be created for each of the modules to include the necessary information for the module and each feature it contains. All these files need to be organized into a file hierarchy that matches Figure 4.2 and placed in the */data* directory on the spacecraft for the monitoring library to use.

4.5 Monitoring Library

The monitoring module that is to run on the spacecraft was designed to fulfill all the requirements listed in Section 4.1. The code to perform the monitoring should be compatible with the flight software and integrated in way that makes full use of the libraries and abstractions provided. The first instinct was to implement the health monitoring system as a new process on the spacecraft. This idea made sense because monitoring can be seen as its own entity in the modular design of the architecture, and would therefore fit well. It would interact with other processes to obtain information to do the monitoring and would exist in its own code-space. However, this design doesn't fit well with the requirement of making this system generic. The hardware components and complexity of each spacecraft differ from mission to mission and this would require frequent updates to the code of the process to make the system compatible to the new hardware of a different mission.

To better fit this requirement, the monitoring phase is provided as a new library that processes can import for monitoring. Through this design, each process can have a unique monitoring module running with its own parameters and feature sets. Also, the library can be easily and separately updated to coincide with any changes to hardware components without any major changes needed in the system process itself. The library will initialize the data structure objects containing the information for monitoring, and have methods for checking anomalies and cleanup.

At a high level, the monitoring library will add scheduled events to each importing process so that the monitoring can run continuously. The configuration and cluster files will be read to set up the data structures needed to perform monitoring. An anomaly checking event will send commands to obtain the values for each feature in the module. Once all the data has arrived, the monitoring algorithm will return a deviation score that is reported. This cycle will continue throughout the mission life cycle.

4.5.1 Add scheduled Events

Since the monitoring is designed as a library, its functions for setup, processing, updating, and cleanup need to be called by each process where the monitoring will run. The initialization needs to only run once at the beginning of each process to setup the data structures that will be used. However, the anomaly detection and update functions are events that need to run continuously throughout the mission cycle . As a result, these events need to be registered to run at given time intervals. This is where the event handling library is used to register these events that will continuously run on the process's event queue.

4.5.2 UDP commands response

Real time values for the features in each module are required for the monitoring algorithm to process and return a deviation score. Each process owns a set of sensors that it can read and obtain a value from easily. The values for almost all the sensors owned by a process can be obtained through UDP commands that the ground or other processes send. The response packets can then be parsed for the individual feature values required. We utilize this capability to acquire all the values needed in each module to perform the monitoring.

This library can be utilized by any process in the system. Each feature in a module will have a command number and process name that is used to generate and send these commands to the correct destination command handler. The command handler will generate the response, and it will be sent back to the original process. To improve the speed and efficiency of the system, multiple feature values can be obtained from one call if their corresponding command numbers match. This process will be discussed more in depth in the implementation section.

4.5.3 API

The API for this library consists of four simple methods:

1. **hms_init -** This function performs the initialization of the data structures that provide information for the monitoring. This initialization step consists of allocating proper amounts of memory for the data structures, reading the cluster and configuration files, storing this information, and returning a *HMSData* object consisting of all necessary data fields to be used by the other methods.

2. **hms_check_anomaly -** This function uses the *HMSData* object created in the initialization step to perform the monitoring and report any anomalies. It sends all commands needed to obtain feature values, and then runs the IMS monitoring algorithm to return a deviation score. This score is compared against a threshold and an appropriate health response is generated.

3. **hms_check_file_modification -** This function checks for any updates to configuration or cluster file that may have occurred. If a change is detected, all relevant models will be updated.

4. **hms_cleanup -** Standard cleanup function to stop the monitoring and free up any allocated memory

4.5.4 Error reporting

This system will report any anomalies when the deviation score calculated by *hms_check_anomaly* is greater than the threshold value given. Once an anomaly is detected, it will be reported by the beacon packets that are broadcast by the satellite. In addition to this, the timestamp, deviation value and the individual feature error contributions will be stored in a file on the *AnomOutput* directory in Figure 4.2.

In response to an anomaly, the system could also respond by increasing the rate of telemetry storage in the database so a more thorough investigation of the cause can be done.

4.6 Adaptability

Finally the system must be adaptable throughout its mission life cycle. Usually to perform updates on the satellite, a new command is generated that provides the update functionality. Since we are using a library to perform monitoring, creating a command for each process that uses the library is bad design.

The update functionality for the monitoring system will utilize the file hierarchy design in Figure 4.2. The basic concept to update the models on the system is to copy the updated file to its corresponding model sub-directory in either the clusterFile or configuration directory. Files can be sent to the satellite while it is in orbit through its IP address. The **hms_check_file_modification** will run at a set interval and check for changes in the directory structure. If a change is detected, it will reconfigure the models to include the changes. In the case where the model update fails, it will delete the new file and revert back to the original. It is for this reason, that each model has its own directory.

The system can also be configured to stop running if it is performing poorly. This can simply be done by changing a value in the HMSRun.txt file.

Chapter 5

IMPLEMENTATION

This section discusses the implementation of the PolySat health monitoring system based on the system design presented in the last section. It does a more detailed analysis on how the learning module works and what its output looks like. The monitoring library's integration into the current flight software is also examined in more depth.

5.1 Learning

The learning phase which runs on the ground was written in Python to take advantage of its ease of development and plethora of libraries and frameworks. This includes machine learning libraries such as Sci-Kit learn and scientific computation and statistical libraries such as NumPy. NumPy was a very useful library because it provided the ability to create homogeneous multi-dimensional arrays in Python, and use many of its built in high-level mathematical and statistical functions to manipulate these arrays. These arrays are much faster and efficient to do scientific computation on then Python's built in lists.

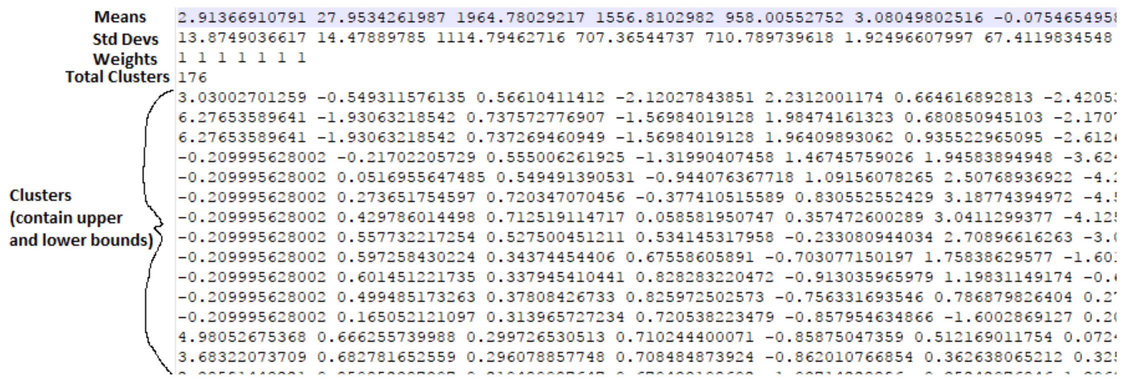
The learning program that was created is a single file that executes one training module with a unique feature set at a time and outputs the finished cluster file. If there were multiple modules created, this program needs to be run multiple times with the parameters for each module provided as input.

The implementation of the learning algorithm is very similar to the one mentioned in Section 2.3.1. There exists a global Cluster database object that contains all clusters that are generated by the algorithm for each module. A Cluster class also exists which constructs a Cluster object that contains the upper and lower bound vectors, number of elements, and the centroid of the cluster. This Cluster object also contains various function that can add, remove, or manipulate data vectors in the cluster.

As input, this program takes a list of file names that contain the time series data required to train the model and generate the Cluster database. It also takes the threshold value ϵ that specifies the cluster radius. Since the training data can possibly contain hundreds of columns worth of feature values, a list of column numbers that specify which features constitute the module needs to be supplied. The configuration file can be referenced for the feature names and the corresponding columns found in the training data. Finally, a list of weights for each feature is required if you want to scale certain features more than others.

Once it obtains all necessary input, the program parses through each file and generates a NumPy array for each data vector. The end result of this parsing is a multi-dimensional array that contains all the data vectors that were provided by the training data. Since the training data may contain portions where some of the data is not available for a subset of the sensors, there is some cleanup performed to remove such vectors. This array of data vectors is then scaled using z-score normalization and the weights that were provided as input. After this, each scaled data vector is processed as described in Section 2.3.1, and the final product is a list of Clusters in the Cluster database object that defines the model for the selected feature set module.

The content of the Cluster database is then written to a file with the name of the model. This file consists of floating point numbers which represent the upper and lower bounds for each cluster. Each cluster is defined in one line with the upper-bounds representing the first half, and the lower-bounds representing the last half. Since any new monitoring input vector needs to be scaled according to the training data, scaling information is also required in the file. For the normalization, the means and standard deviation of the features are added. The weights of each feature are also saved along with the total number of clusters in the file so the parsing in the monitoring library knows how many lines to read. The content of this file is shown in Figure 5.1. This process is then repeated for each monitoring module defined by the configuration files.



```
2.91366910791 27.9534261987 1964.78029217 1556.8102982 958.00552752 3.08049802516 -0.07546549
13.8749036617 14.47889785 1114.79462716 707.36544737 710.789739618 1.92496607997 67.4119834548
1 1 1 1 1 1 1
176
3.03002701259 -0.549311576135 0.56610411412 -2.12027843851 2.2312001174 0.664616892813 -2.4205
6.27653589641 -1.93063218542 0.737572776907 -1.56984019128 1.98474161323 0.680850945103 -2.170
6.27653589641 -1.93063218542 0.737269460949 -1.56984019128 1.96409893062 0.935522965095 -2.612
-0.209995628002 -0.21702205729 0.555006261925 -1.31990407458 1.46745759026 1.94583894948 -3.62
-0.209995628002 0.0516955647485 0.549491390531 -0.944076367718 1.09156078265 2.50768936922 -4.
-0.209995628002 0.273651754597 0.720347070456 -0.377410515589 0.830552552429 3.18774394972 -4.
-0.209995628002 0.429786014498 0.712519114717 0.058581950747 0.357472600289 3.0411299377 -4.12
-0.209995628002 0.557732217254 0.527500451211 0.534145317958 -0.233080944034 2.70896616263 -3.
-0.209995628002 0.597258430224 0.34374454406 0.67558605891 -0.703077150197 1.75838629577 -1.60
-0.209995628002 0.601451221735 0.337945410441 0.828283220472 -0.913035965979 1.19831149174 -0.
-0.209995628002 0.499485173263 0.37808426733 0.825972502573 -0.756331693546 0.786879826404 0.2
-0.209995628002 0.165052121097 0.313965727234 0.720538223479 -0.857954634866 -1.6002869127 0.2
4.98052675368 0.666255739988 0.299726530513 0.710244400071 -0.85875047359 0.512169011754 0.072
3.68322073709 0.682781652559 0.296078857748 0.708484873924 -0.862010766854 0.362638065212 0.32
```

Figure 5.1: Example of a cluster file

The cluster files and configuration files are then saved in the directory structure show in Figure 4.2 and uploaded to the file system on the satellite.

5.2 Monitoring - Initializing HMS Object

Like most of the code written in the flight software, the monitoring library uses the C programming language. Many existing flight software libraries use object-oriented design choices to provide their functionality. Since C doesn't provide a built-in mechanism for object-oriented programming, it is mimicked. In these cases, a data object in the form of a C struct is created that encompasses all of the data structures and variables necessary for that library to work. This data object is then passed as a parameter to most library functions for use or manipulation. A similar design choice was made for the health monitoring library. This library starts by initializing a *HMSData* object which contains all monitoring information and is passed to the other functions provided by the library. The *HMSData* object has references to multiple other *HMSModule* objects which define a monitoring module. This includes information such as the model name, features, current state, and feature values which make up the module. It also contains a reference to a *ClusterState* object, which contains the cluster file information including the means, standard deviations, weights, and actual clusters. This overall structure is shown in Figure 5.2.

The *HMSData* object is represented as a C struct and some of its important fields include:

- **procName -** The process's name which is creating the object for monitoring.
- **numModules -** The number of monitoring modules which are to run for the given process name.
- **udpPacketProcs -** This data structure will hold all UDP commands that need to be sent to each process to obtain all data vector values. This information is provided by the configuration file, where each feature has a corresponding process name and command number which defines where to obtain its value. Since a large portion of data vector values can be obtained from the same command, this structure contains only one reference to each required command so there are no repeats.
- **filesInfo -** This data structure contains the file paths for the latest versions of the configuration and cluster files for each module.
- **modules** - A reference to a list of *HMSModule* objects.

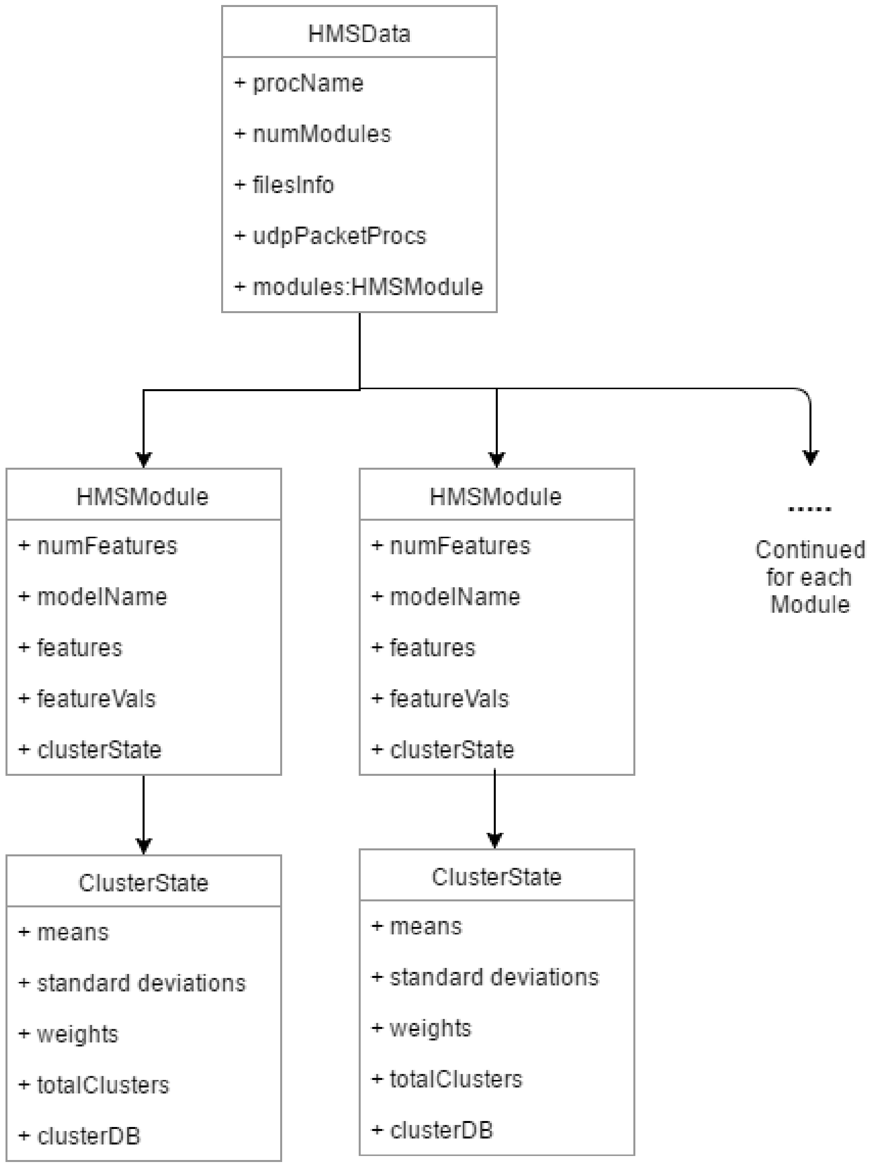


Figure 5.2: Object structure for monitoring library

The *HMSModule* object's important fields include:

- **numFeatures -** The number of features that exist for the the given module.
- **features** - A data structure that holds important information for all the features in the module. For each feature , this includes the name of the sensor or telemetry point, the process which owns it, its command and response number, and a function pointer to a function that extracts the feature value from the

response command packet.

- **modelName -** The unique name given to the module, which corresponds to the correct directory in the file hierarchy.
- **featureVals -** An array that holds the feature values from the commands that have been sent and acts as the input vector. This array has a size equivalent to *numFeatures* and is reset every time a new check anomaly event is fired. Processing of the monitoring vector does not occur until this array is full. A count variable is also provided that is incremented every time a data value has been received. This count is initialized to zero at the beginning of each monitoring event.
- **clusterState -** A reference to the ClusterState object which contains the means, standards devations, weights, and clusters that correspond to the given model. This object is used by the monitoring algorithm to compare the new input vector to the cluster knowledge base.

Now that we have gone over the structure and important fields of the monitoring objects, the initialization process will be discussed.

5.2.1 Read Files from directory

The first step involves dynamically allocating the space for the monitoring objects and initializing the fields to zero or NULL. Next, the file paths for the latest version of the configuration and cluster files need to be placed within the filesInfo field of the *HMSData* object so that we can process each file. This is accomplished by calling a function that returns a reference to an object that contains these paths. This function starts by looking in the *ModuleCnfgFiles* directory and going through each Model sub-directory within it. For each Model subdirectory, it looks at each file and returns the path of the one that has the latest last modified date. The last modified date is also saved as part of this object. Since we want to work with the latest models, this ensures that we are running the monitoring with the most up to date files. A similar process is repeated for the files in the *ClusterFiles* directory and the paths to the latest cluster files are returned.

5.2.2 Read Module Cnfg File

Next, we loop through each module configuration file contained in the filesInfo data structure. Since this object has a path of every model in the system, we need to extract only the one's that were defined for the current process that the library is being run on. Each configuration file is read and parsed using the built in library that handles configuration files. This library returns an object that has all of the tags and values in the file. The process name field in the config file is compared against the current process name to determine if that module belongs to the process. If it is a match, then a new *HMSModule* object is initialized and added to the *HMSData* object. The rest of the fields in the *HMSModule* object are initialized using the information that was contained in the configuration file. The *featureVals* array is allocated with space for *numFeatures* and is initialized to have default values of zero.

5.2.3 Register Feature callbacks

The next step involves setting up the *features* data structure with the feature information from the configuration file.

The library includes a large array of static structs which contains all possible sensors/features that exist for a given satellite. Each struct within this array contain the same fields as the *features* field in the *HMSModule* object. One of the most important fields in this struct is the function pointer which for a provided response packet, extracts the specific feature value. This function calculates the offset of the specific field in the response packet and reads the correct number of bytes that correspond to the size of the feature. It also does any unit conversions required such as changing a raw temperature sensor value to degrees Celsius. It is important for the units of each feature to match the units of the training cluster files so there exists no discrepancies which may end up with bad results. In most cases, we use the raw sensor values provided and perform no conversions to reduce some complexity. The training data must also have its data stored as raw values returned by the sensors.

This large array of structs is traversed and a reference to the matching feature is returned to be saved in the *features* field. The command and response number of the handler that contains the specific feature value are also provided as members of this object. Usually, a command handler that returns telemetry status contains many telemetry fields, one for each sensor the process owns. This results in multiple features requiring the same command to obtain their values. Instead of sending commands the naive way where a new command is sent for each feature, features which share commands will all get their values from one copy of the command. To ensure this happens, any feature that requires a command that hasn't been seen before will save the command and response number in the *udpPacketProcs* field in the *HMSData* object. By the end of the initialization, this field will have one copy of all commands that need to be sent to obtain feature values for all the modules. This design choice significantly improves the performance and efficiency of the system.

Now any time a command response arrives, all feature values can be easily extracted into the *featureVals* array which is used in further processing.

5.2.4 Read Cluster Files

The last step in the initialization function involves reading the cluster file that corresponds to the model name. Once again the *filesInfo* object is traversed to find the file path for the latest version of the cluster file for the specified model name. Once the correct file is found, a function parses the cluster file, allocates necessary variables, and saves the means, standard deviations, weights, and clusters in the *clusterState* object. The parser is smart enough to separate the single line that represents each cluster to the correct upper and lower bounds for the cluster. This object contains all the information required for the monitoring algorithm to produce a deviation value for a given input vector. This finishes the initialization for one module.

The initialization of the *HMSData* object is complete when this function has looped through each module file that belongs to the given process. This object is then passed to the check anomaly, check for updates, and clean up functions that are provided as part of the API. A summary of this is illustrated in Figure 5.3.

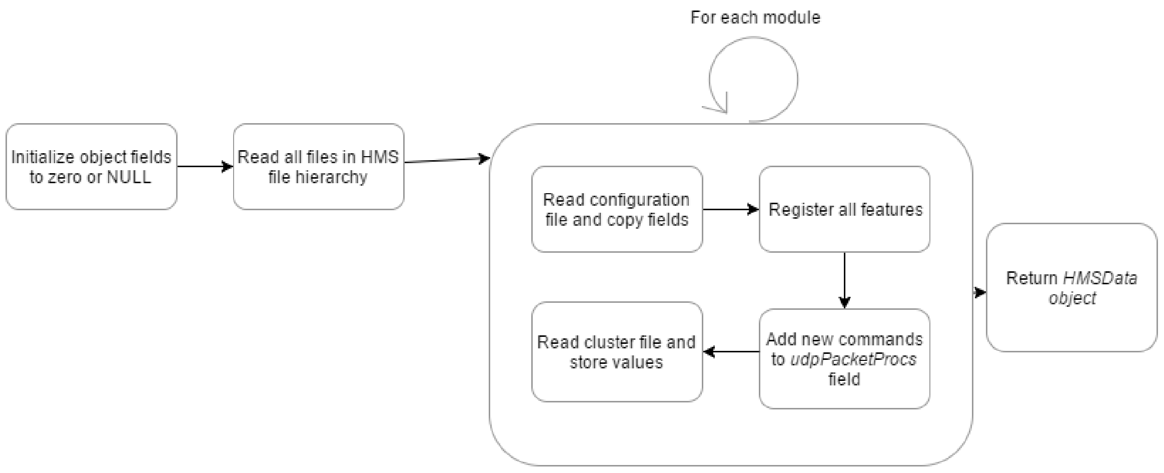

For each module
Initialize object fields to zero or NULL
Read all files in HMS file hierarchy
Read configuration file and copy fields
Register all features
Add new commands to udpPacketProcs field
Read cluster file and store values
Return HMSData object

Figure 5.3: Flow of the initialization process

5.3 Check Anomaly

Once initialization is complete, monitoring of new input vectors can begin. The sample rate for monitoring can vary and is expressed by the time period given to the event handler which calls the monitoring function. To get high sample rate real-time monitoring, the time given would be very small. This does come at the cost of more processing power as the monitoring would continuously run at a high rate. A high sample rate is also not efficient if the response values don't change very much. To balance efficiency and performance, a small sample rate of one run every 30 seconds was chosen. This rate seemed to provide the best monitoring capability.

The check anomaly function sets and processes all the modules defined for the process at the same time, instead of doing it individually. This is more efficient in that the commands sent can set the values for features across all modules rather than for the individual module. This results in less commands being sent.

The check anomaly function takes as input the *HMSData* object, a *ProcessData* object, and the threshold value determines if a non-zero deviation score is accepted as nominal. The *ProcessData* object is created for each process in the system and holds important state and event handling information. This object will be used for some event handling described later. This function first saves a reference to the *HMSData* object that was passed in a global variable. This is necessary for the response command handler to know which object to reference and save the feature values. The next step involves setting all the fields in the *featureVals* array and the count variable to zero to set up a new input vector.

5.3.1 Send all commands

Once the new input vector is set, all the commands to set the feature values need to be sent to the correct processes to obtain response packets with the values. To do this, all the commands in the *udpPacketProcs* field are sent. Once a command is sent, the response packet is sent as another command by the supplying process. We need to be able to process this response. To do this, a command handler needs to be set for the given response command. The commanding library provides a set of command handlers for the default commands set for any given process. These command handlers call a callback function which processes each request. The command library also provides a way to set a new handler for any command number specified. This method is usually used when a process does inter-process communication and needs to do something with the response. We require a similar sort of functionality for the commands sent by this library. We need to set a new handler that allows us to access the response packet. However, since this is a library, we don't want to overwrite any other command handlers that may have been set by the process itself to perform some important function. In order to solve this problem, a new network socket needs to be created for each command that handles both the sending of the command and the response.

To add this functionality, some additions needed to be made in the existing commanding library to give the ability to send and receive commands on a new socket. A new function was added that does this very thing. It takes as input a callback function that acts as the response handler for the given command, and the *ProcessData* object. The function creates the new socket, and sets up a sockaddr_in struct so that the command can be sent to the correct destination. The event handler from the *ProcessData* object is then used to set a new file descriptor event when a response has been sent back. This event will call a callback function that reads the response from the socket and calls the given response handler. A timeout is also set in case the command fails. Once this setup is complete the command is sent. These steps are summarized in Figure 5.4

This process repeats for each command that was defined in the *udpPacketProcs* data structure.

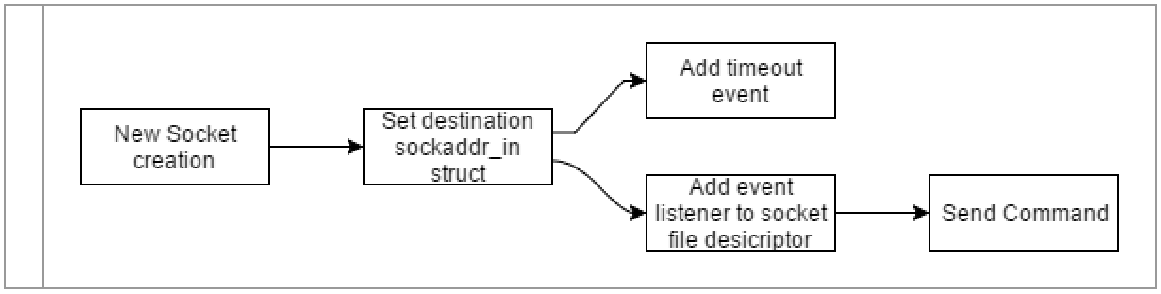

New Socket creation
Set destination sockaddr_in struct
Add timeout event
Add event listener to socket file desicriptor
Send Command

Figure 5.4: Steps for sending a command from the monitoring library

5.3.2 Response handlers

When the command response arrives on the socket file descriptor, the socket is read and the event callback function calls the response handler that was initially passed to the new commanding function. This is illustrated in Figure 5.5. These response handlers extract all feature values from the response packet and perform any further processing. A response handler needs to be created for each process since process's command numbers can overlap. A new handler for each process ensures we know the source and structure of the response.

The response handler first extracts the relevant feature values from the response packets. It does this by looping through each module and looking for features that match the response process name and command response number. Once there is a

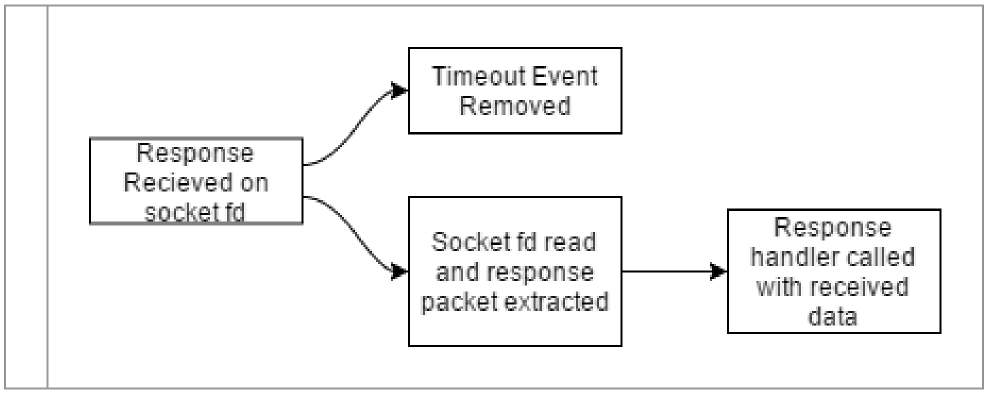

Timeout Event Removed
Response Recieved on socket fd
Socket fd read and response packet extracted
Response handler called with received data

Figure 5.5: Response handler flow

match, the response packet is sent as a parameter to the feature's function pointer so that it can get the value. This value is saved in the correct spot of the *featureVals* array for the module the feature belongs to. The feature set count variable is also incremented as each value is set. By the end of this, all features that had their values stored in this response packet will have their values set.

Lastly, the response handler checks if every feature has been set so that the monitoring algorithm can be called. It does this by once again looping through each module and checking if the feature's set count variable equals the total number of features in the module. If this is the case, then the *featureVals* array for each module is set and can be processed. If not, then all command responses have not yet been returned and the function returns.

5.3.3 Call monitoring algorithm

Once all features values have been set, the IMS monitoring algorithm described in Section 2.3.2 can be called on each module feature set. The algorithm takes as input the new monitoring vector along with the *ClusterState* object which contains the cluster knowledge base for the module. Each monitoring vector is normalized and scaled accordingly to the means, standard deviations, and weights that were provided by the cluster file. Once normalized, the steps described in Section 2.3.2 are taken to produce a measure of how close the monitoring vector behaves with the nominal model.

5.3.4 Deviation score

The monitoring algorithm returns a deviation value that specifies the distance between the input vector and the closest nominal cluster. If the deviation value is zero, then the input vector exists within a defined cluster and is accepted as nominal. If it is a non-zero number, then it is compared against the threshold value that was given as a parameter. If it is lower than this threshold, it will be accepted as nominal. If not, then it will be rejected and an anomaly will be reported. Along with the deviation score, the monitoring algorithm saves the individual deviation sums that each feature contributed to the overall deviation score. This allows the operator analyzing the error to identify which features are causing the issue.

When there is an anomaly detected for a certain module, a time-stamp, deviation score, and individual sums are saved to a file in the *AnomOutput* subdirectory of the file hierarchy in figure 4.2. This is the file that is read by the operator to analyze the issues. Along with this error report, the beacon can broadcast that error has occurred. These steps are illustrated in Figure 5.6.

5.4 Updating model

The ability to update the monitoring models during the mission cycle is an important requirement. As discussed in the overview section, updating the models involves adding, modifying, or deleting files in the IMS file hierarchy. This is done by using a custom File Transfer Protocol on the satellite which handles the transferring of files while it is in orbit. The transfered files need to be placed in the appropriate

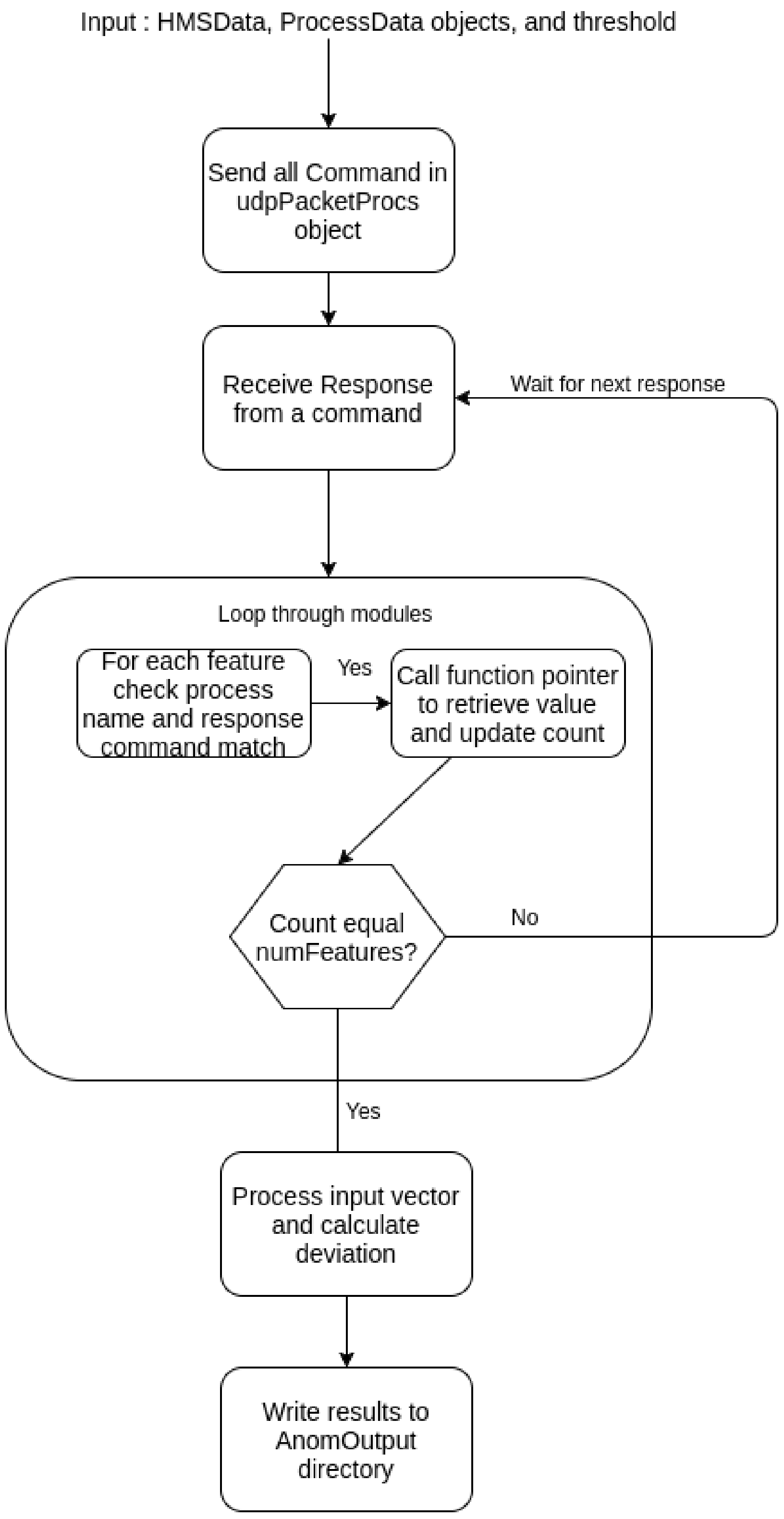

Input : HMSData, ProcessData objects, and threshold
Send all Command in udpPacketProcs object
Receive Response from a command
Wait for next response
Loop through modules
For each feature check process name and response command match
Yes
Call function pointer to retrieve value and update count
Count equal numFeatures?
No
Yes
Process input vector and calculate deviation
Write results to AnomOutput directory

Figure 5.6: Flow of the check anomaly function

directories. The monitoring library provides a function *hms_check_file_modification* that detects changes by setting up an event that periodically checks the content of the directory.

This function takes as input the *HMSData* object and starts by calling the function that returns the latest versions of the configuration and cluster files in the file hierarchy which was discussed in Section 5.2.1. The last modified date of each file that was returned is compared against its counterpart in the *filesInfo* field of the *HMSData* object which contains a reference to all files that are currently being used in monitoring. If there is a discrepancy in the number of files or the last modified date for a certain model, an update is detected.

Once an update is detected, a copy of the old *filesInfo* object is saved. The new updated files object then replaces the content in *filesInfo*, and the *HMSData* object is reinitialized to reflect the new changes. There can be errors that occur during re-initialization that may be caused by human error in the updated files or during the transferring process. In these cases, we don't want the system to completely break, so the reference to the old *filesInfo* object is used to reinitialize with the previous files that are known to work. The function also deletes the faulty files so when the event fires again in the future, the same problem won't occur. This design provides a powerful way to update any cluster or configuration file for monitoring without the need to add new commands.

5.4.1 Stopping HMS

Stopping the health monitoring system at any time is also an important ability that the library provides. In cases where the library is performing poorly, which can be caused if the models were generated using poor training data, the monitoring should stop so unnecessary resources aren't used by the system. The *HMSRun.txt* file that exists in the root of the file hierarchy allows us to stop and restart the monitoring by simply changing the one byte flag that it contains. If the operator wants to stop the monitoring, the content of the file is changed to 0. If a restart is required, the content is changed back to a non-zero number. This file is checked as part of the *hms_check_file_modification* function and the value saved in the data object. Every time the check anomaly event is triggered, it checks if the flag is set to continue processing.

5.5 Cleanup

A cleanup function is also provided to free any allocated memory and gracefully end the monitoring in case there is some error, or the operator wants to stop the system.

5.6 Adding Events to Selected Processes

Any process that has its own instance of the monitoring library needs to call the API to initialize and run the monitoring. The initialization occurs once in the process's main event loop. Once successful, events are registered for the *check_anomaly* and *hms_check_file_modification* functions to run at certain intervals. As discussed before, the check anomaly function runs once every 30 seconds. Updates to the monitoring files should be rare and far in between. As a result, the update function is registered to run every 30 minutes. Once these events are registered in the process event loop, the monitoring library can provide its functionality.

Overall, the system was designed and implemented to meet the requirements discussed in Section 4.1. This results in a robust and efficient system that performs very well.

Chapter 6

VALIDATION AND RESULTS

The integration of the new Inductive Health Monitoring System into the PolySat flight software code base will have to be thoroughly tested and validated to ensure that the system provides good results and doesn't waste any precious resources on the CubeSat. The system should at least be able to detect major failures that occur with the hope that it picks up subtle errors as well. Testing the system involves validating the implementation of the IMS algorithms, checking if the system can identify anomalies on archived events, running the system real-time on a test unit, and checking resource consumption. This validation and experimental testing is detailed in this chapter and the results are analyzed.

6.1 Algorithm Validation

It is important to ensure that the IMS learning and monitoring algorithms are working properly because the success of the system is highly dependent on these properly identifying anomalies. They make up the core of the IMS system and a small problem in one may cause incorrect results and wrong predictions by the system.

6.1.1 Learning module clustering

The first step in testing the IMS learning algorithm was to examine the clusters that were being generated from various data. Correct cluster formation and hyper-cube generation is vital for the correct modeling of the nominal operating regions of the satellite, and for the monitoring algorithm to provide accurate results.

In order to validate the IMS clustering approach, multiple two dimensional data sets were generated by the *make_blobs* function in the Sci-Kit learn library provided in Python. Two-dimensions were chosen so that it would be easier to visualize and examine the clusters that were being formed. The *make_blobs* function creates N number of random samples that are clustered around a given number of centers X. A cluster standard deviation σ is also given to control the spread of the points. Multiple data sets with different values for N, X, and σ were created and tested against the IMS clustering. The IMS clustering algorithm was also given different threshold values to see how it would effect the clusters formed.

Two examples of the results are show in Figures 6.1 and 6.2. The data generated in Figure 6.1 had 5000 samples clustered around 6 centers. The IMS clustering algorithm ran with a threshold of 0.75 and produced the clusters shown. The data points in each cluster formed by the algorithm have a different color, and the bounding rectangle the represents the cluster is displayed around the points. This rectangle defines the upper and lower ranges of the cluster in each dimension. As can be seen, the algorithm did a good job finding the points in all six clusters. Since the dataset had clusters that varied in size, some overlap between clusters can be seen. This is expected due to the fact that the size of the cluster is highly dependent on the threshold value given. If the threshold is large, then it may capture some points from a nearby cluster that it may not belong too. Some of the cluster overlap is also due to the fact that the points are randomly created around the cluster, and not in time series. Time series data creates nicely defined clusters that have minimal overlap.

This result also demonstrates how large threshold values can cause the operating region to be too generalized. Since IMS doesn't use a more traditional clustering approach where the bounds of the cluster will be tightly defined around its data points, it could accept vectors that may not belong. Since the clusters generated by the dataset are more spherical, empty space exists in the rectangular clusters generated by IMS. This is especially apparent in the cluster that contains the blue points in Figure 6.1. If an input vector was chosen that exists on the lower right corner of the cluster, it would consider it nominal since it is within the bounds. However, we can clearly see how this point should not belong and is more of an outlier compared to the other data points in the cluster. This again shows the important of the threshold value. In this case the threshold value was a bit too high for this data and lead to a lot of empty space being captured in the cluster. If you want a more higher fidelity model of the data that has tighter bounds and fits the data better, you may want to choose smaller threshold value.

The data generated in Figure 6.2 had 10 cluster centers with the samples creating smaller clusters. IMS ran on this data set with a lower threshold value of 0.5 and produced the following clusters. IMS once again did a good job finding the clusters and defining the bounding rectangle. Once again there is overlap, but this time it is mostly due to the threshold value being too small which split certain larger clusters into two. It is important to note that these overlaps won't result in bad monitoring prediction due to the fact that the application isn't classification where precise clusters are necessary. As long as the behavior is captured in any cluster, monitoring should be fine.

These good results carry on into higher dimensional data as well. The clusters generated with a N-dimensional data set would create a N-dimensional hyper-cube around the cluster which is hard to illustrate in this case. A two-dimensional test allowed us to verify that the clustering was working properly.

6.1.2 Monitoring module

The IMS monitoring algorithm was also tested to verify that it would correctly identify any anomalous results and produce a good deviation value. This algorithm would take input data vectors and use the cluster knowledge base to predict whether

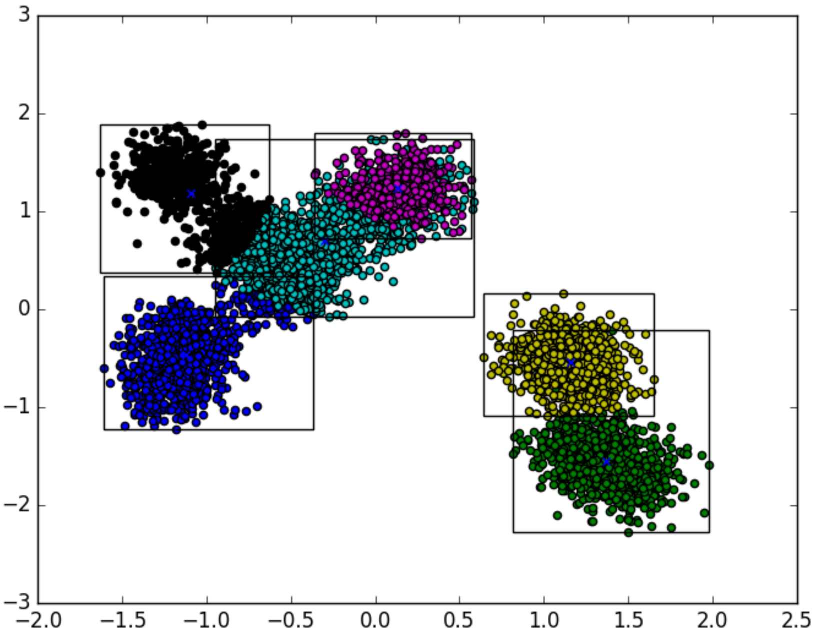


Figure 6.1: Cluster formation, N=5000, threshold = .75, X = 6

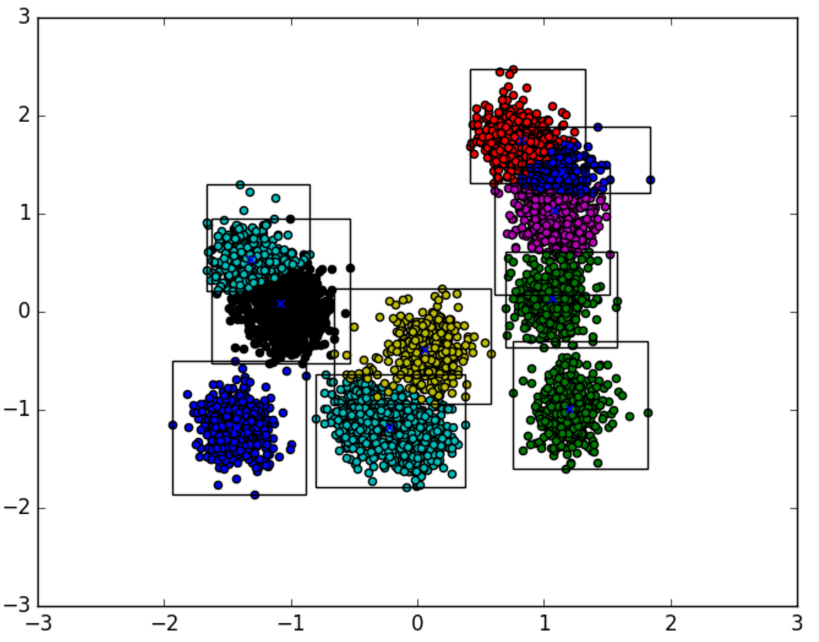


Figure 6.2: Cluster formation, N=5000, threshold = .5, X = 10

or not the system is performing nominally. To perform a preliminary test of this, the clusters generated in Figures 6.1 and 6.2 were used as the knowledge base. Since the bounding rectangles were clearly visible, it was easy to pick a point inside or outside any defined cluster. After picking multiple points, the monitoring algorithm would correctly identify if the point chosen was anomalous or not, verifying that the monitoring works in this simple case.

The monitoring was also tested on a larger dataset with multiple features. David Iverson, the original creator of IMS, provided some example data sets that he tested his implementation against. This data set was from a real application that involved collecting telemetry from a rocket engine fire test. He provided a nominal dataset with over 150,000 vectors consisting of seven features that measured the pressure of various sensors to train against. He also provided a test data set that had some known anomalies inserted.

Our implementation of IMS was trained against the training data set given, and monitoring was run on the test set. The monitoring algorithm would read every input vector from the test set file, normalize it, and process it using the knowledge base to produce a deviation score for each vector. The result was written to a file for analysis. The results of the run were plotted in Figure 6.3. Iverson also provided a graph of his results for reference as shown in Figure 6.4. The first 3/4th of the test data set had nominal data vectors as seen by the zero deviation score for these points. In the results of our run and Iverson's, the first anomaly is detected at the same time of about 52644.06718 GMT. The rest of the data has deviation scores that increase exponentially at first and then slow down and grow polynomially. Looking at both Figure's 6.3 and 6.4, similar trends can be seen highlighting that our implementation works as expected. Iverson's graph looks slightly different because it's exponential growth is shorter and it produces a much smoother curve. These differences are likely due to different threshold value's given in each run. His learning module created 236 clusters while ours only generated 72 which leads to different deviation scores being generated. There may have also been slight differences in the implementation and the way the deviation score is calculated leading to these slightly different results. The important point is that our run was successful in identifying anomalous points and produced growing deviation scores.

The validation of our implementation of the learning and monitoring algorithms went very well and produced results that were expected. The system was able to identify nominal and anomalous results and this gave us good confidence that these algorithms would work correctly for our application on CubeSat's.

6.2 Experimental Tests on archived data

The next phase of testing was to utilize real CubeSat flight data with known anomalies to see if the system could identify the problems. Archived flight data from the Intelligent Payload Experiment (IPEX) described in Section 4.4 was used for this purpose. The original IPEX flight dataset contained anomalies that occurred to the satellite during the mission. There were two major problems that the data reflected. To create a valid nominal training data set, the sensors that were contributing to these anomalies were manually corrected to reflect nominal behavior. This training set would be used for training the system and creating the knowledge base, while the original data set would be used as the test set for monitoring. The goal was to have the system identify the anomalous sensors.

As discussed in Section 4.4, we used a correlation matrix on the IPEX training set to generate our feature sets or modules for the IMS system to run against. We produced two modules, one that contained most of the temperature sensors in the data, and another which contained the power sensors. See Appendix A for a detailed look at these sensors. To identify which sensors/features were causing the issue, the

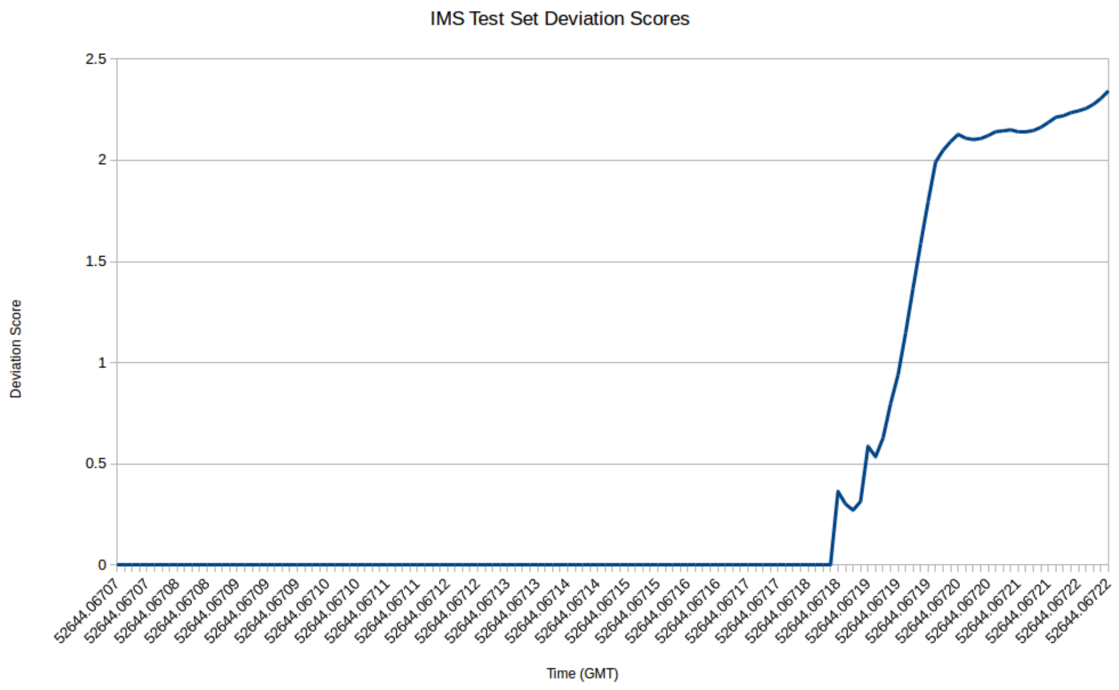


Figure 6.3: Deviation scores produced by running IMS on test data set using our implementation

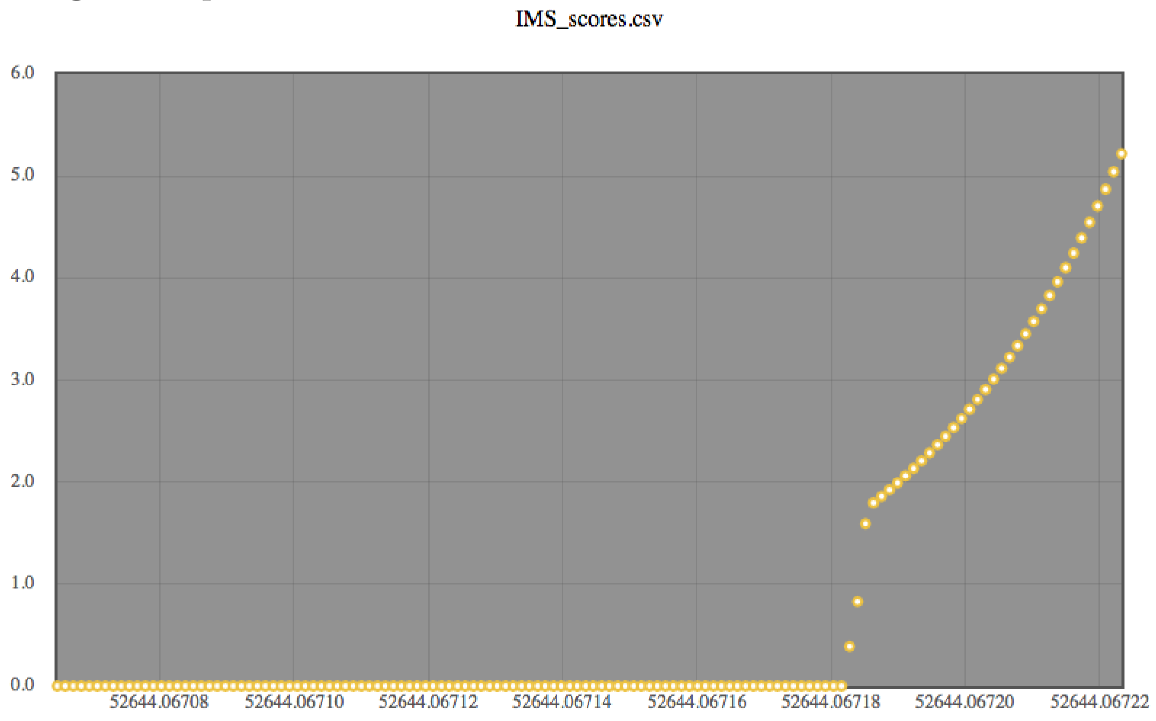


Figure 6.4: Deviation scores produced by IMS given by Iverson

individual sum that each feature contributed to the overall deviation score was also stored. These modules were run separately and the deviation score along with the individual sums were written to a file to analyze.

6.2.1 Temperature Model

The first run of the monitoring system against the IPEX flight data used the temperature model and produced the individual feature deviation sums seen in Figure 6.5. This graph reflects the overall sum of the error produced by each feature throughout the run and contains a partial set of the features. Looking at this graph, the *boardpx* sensor contributed to the most error. This sensor measures the temperature on one of the side panels of the satellite, in this case the positive X panel that is oriented in the x-axis of the satellites reference frame. Some error sums were also seen in the other side panel temperature sensors, but to a less extent. The other temperature sensors in the model contributed very little to the error sum which indicates that they were performing nominally. Small deviations and sums can mostly be attributed to noise.

Most of the contribution to the error sum was seen in the first quarter of the test set, with very little seen in the latter part. Figure 6.6 shows a line chart of the temperature data for the *boardpx* sensor that was contained in both the training and test set. The red line represents the temperature data for the training set while the blue line is for the test set. Looking at this chart, it can be seen that the temperature of the sensor in the flight set was higher than the seen anywhere in the training set, especially in the beginning. This matches with the large increase in the error sum we saw in the beginning of Figure 6.5.

After looking at these results, it was obvious that there was some sort of anomaly on the *boardpx* side panel. There was in fact a problem with the side panels that occurred during the IPEX mission. The panels were found to not have good thermal

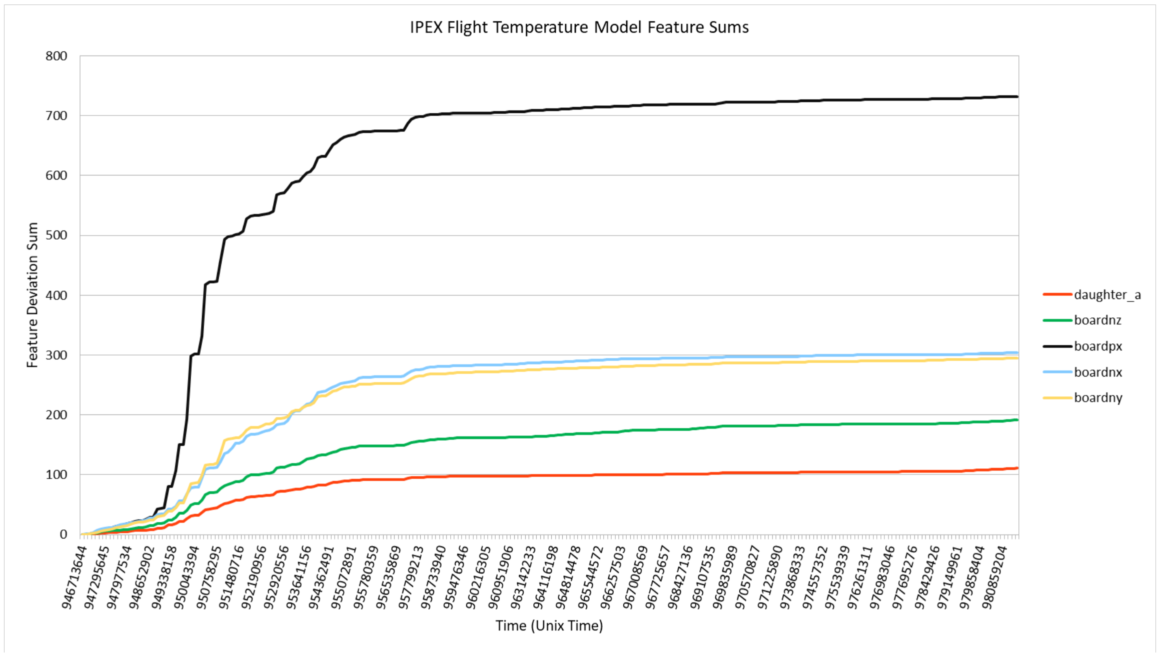


Figure 6.5: The overall sum produced by each feature in IMS run of temperature model

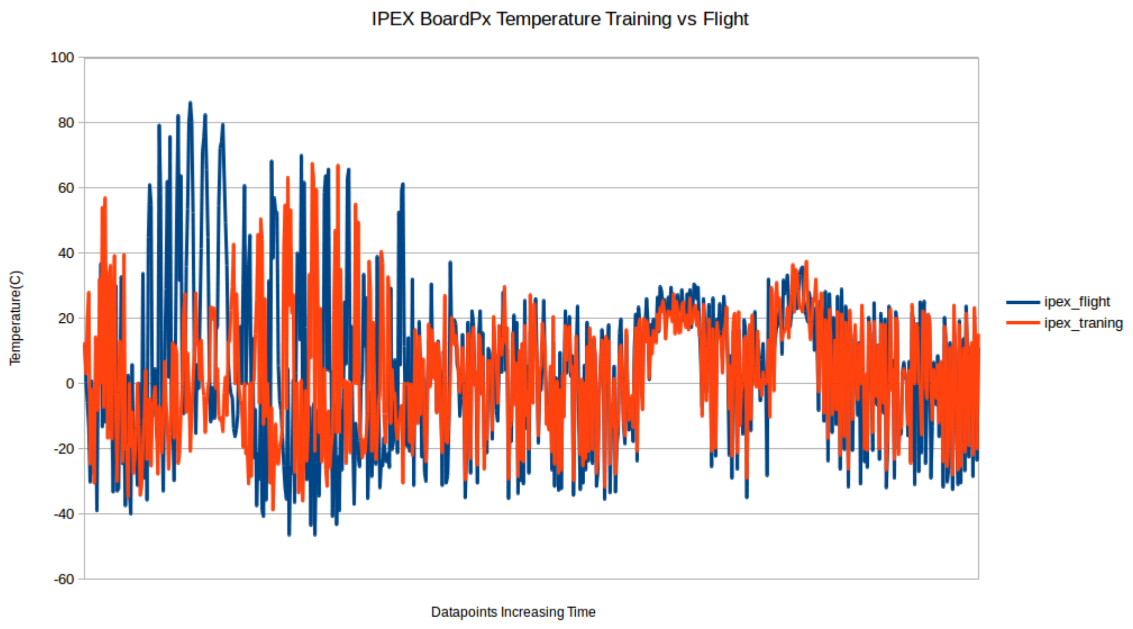


Figure 6.6: Comparison of the temperature data in the training and test data set for the *boardpx* sensor

conductivity and would get too hot when facing the sun. The positive X panel in particular would get very hot as reflected by the sensor readings. The other panels had a brass mass placed behind them that actually absorbed most of the heat, and that is why those sensors contributed much less error. Over time, the thermal conductivity increased and more heat was absorbed as reflected by the flattening of the error sum in Figure 6.5. These results on the temperature model show that our system was able to successfully identify one of the anomalies that occurred during the IPEX mission.

6.2.2 Power Model

The second run of the monitoring system against the IPEX flight data used the power model that contained most of the voltage and current sensors in the data. The results of this run are shown in Figure 6.7 and again reflect the total sum of the error produced by each feature. The top five features that produced the most sum are shown in this chart.

In Figure 6.7, the sensor that produced the most error is the *threeVpl_curr* sensor that measures the current going to the satellite payload co-processor. The second most error is seen by the *threeV_volt* sensor which measures the voltage on the main system board's 3.3 voltage line. There was also smaller amount of error seen in the *threeVpl_volt* sensor that measured the voltage going to the payload. Where as the error for the *threeV_volt* sensor gradually increases, the error on the payload sensor grows rapidly and sees periods of stability. These results indicate that there were some anomalies on the components on which these sensors reside.

There was a confirmed anomaly on the system board's 3.3 voltage line that was measured by the *threeV_volt* sensor. Essentially, there was a hardware defect that caused the voltage on this line to be badly regulated which resulted in higher than expected voltages. This is the reason we saw a steady and continuous increase in the error for that sensor. Our implementation of IMS was therefore successful in identifying this anomaly. However, there were no known anomalies on the threeV payload sensors which these results seem to contradict. Therefore, a false positive was picked up in this run.

At first, the explanation for this behavior was that the training data did not capture the specific relationship between this sensor and the others. This caused the *threeVpl_curr* sensor to contribute the most errors. Also, there is a software telemetry point that specifies when power is switched on to the payload which wasn't found in the training data. If this telemetry point was included, it could have defined distinct nominal operating regions for when the power to the payload was switched on or off which would have led to better results. However, after taking a look again at the correlation matrix, there was very weak correlation found between some of the features in the power model. There were a total of 14 features in this module including the power sensors for the solar panels. We decided to remove and separate some of the solar panel sensors that had weak correlation to another model. After running the monitoring again, we obtained the results in Figure 6.8.

The results in this chart look far better and only contain high error for the problematic *threeV_volt* sensor. The error for the threeV payload power sensors was almost non-existent in this run. This result highlights the need for well formed and highly correlated features in a given model. Large models with a lot of features are good if you want to monitor a lot of sensors. However, they require much more training data to capture every possible behavior between each sensor in the set. If the models contain fewer, highly correlated features, then not as much training data is needed and the results are usually more accurate. Therefore a decision has to be made on the the size of the feature sets that depends on the amount of training data available for use. In our case, it was apparent that more training data was necessary to create a more comprehensive model.

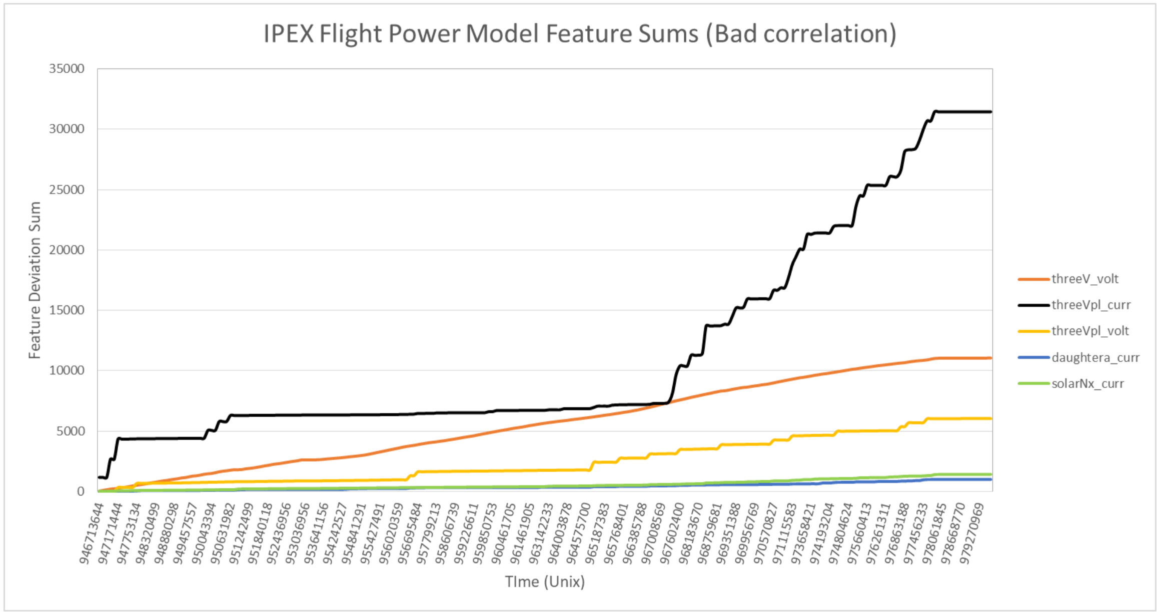


Figure 6.7: The overall sum produced by each feature in IMS run of power model that contained some features with bad correlation

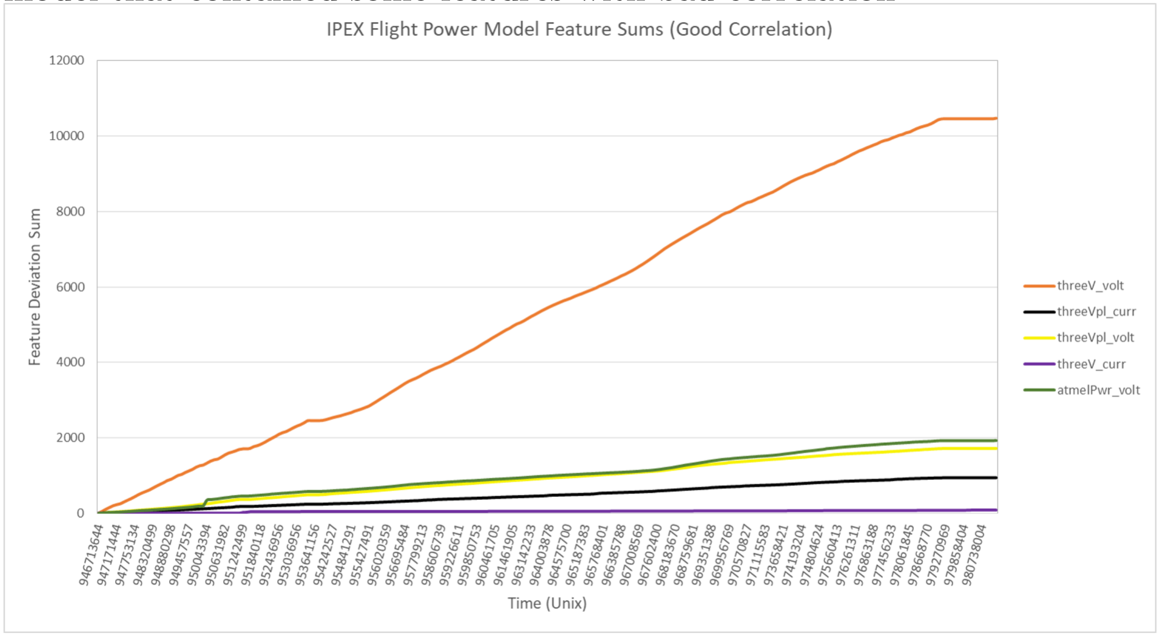


Figure 6.8: The overall sum produced by each feature in IMS run of smaller power model with better correlations

6.3 Real-time CubeSat testing

Through previous testing we were able to conclude that given well formed feature sets, our system worked successfully to identify any anomalies that occurred in the data. The final step involves testing the full IMS system that was integrated into the PolySat flight software to see if we could reliably use this system in future missions. Real time telemetry acquired through commands sent to various processes, were formatted into input vectors and run through the monitoring algorithm as explained in the implementation section.

These tests were run on a experimental test unit for the Ionospheric Scintillation explorer (ISX) mission that is planned to launch in late 2017. During the time of testing, this test unit had most of its components assembled except the side panels which contain the solar panels. This meant that readings from all sensors were not possible, because some of the hardware wasn't connected.

For monitoring, two modules were created that used the feature sets in the temperature and power models that were created using the correlation matrix. To train the model, the archived IPEX data was used. Since the side panels were not attached, the corresponding sensors were removed from the two modules which resulted in the monitoring of most of the sensors that reside on the system board. The temperature module now contained 5 features, and the power module contained 10. These modules were trained and the resulting configuration and cluster files were uploaded to the appropriate directories in the IMS file hierarchy. A temporary process was created in the flight software which initialized the monitoring library objects and set the event for the anomaly detection to run every five seconds.

6.3.1 Temperature Module

The result of running the temperature module on the ISX test unit was very good. The system was run for several minutes and produced a deviation score of zero for every input data vector. Since the IPEX data contains temperatures seen in space, the ambient temperatures in the PolySat lab may have resulted in an anomalous result. However, somewhere in the training flight data, similar temperature conditions to our lab were captured and modeled in the knowledge base which led to nominal results.

The accuracy of this system in orbit is debatable because each mission is in a different orbit. The space environment in different orbits may not be the same which may cause the system to report errors if the temperatures experienced were not in the model. Therefore the system would have to be updated once better telemetry is gathered that is more representative of the space environment.

6.3.2 Power Module

The result of running the power module were not as good as the temperature module. The resulting deviation scores are illustrated in Figure 6.9. As can be seen, the deviation scores were quite high indicating that the system was picking up some sort of anomaly in one or more of the sensors. The error deviation sum for the top contributing features is shown in Figure 6.10. This chart indicates that the Atmel processor current sensor was producing most of the error. Looking at the sensor data in the training data, we found that the average current draw from IPEX was about 35 mA. The standard deviation of this feature in the training data was also very low indicating that this value did not fluctuate much. Looking at the current readings from ISX, we found that the average current draw for this sensor was about 25 mA. This isn't much of a difference and isn't indicative of a major malfunction. This difference can be due to the fact that this is a test unit for a completely different mission, and may have different current draws from the processor. However since the training data does not capture this difference, it resulted in the system reporting a large anomaly for this sensor.

This result once again shows the importance of a good, comprehensive training data set that contains as much component interaction behavior as possible. The more data in the training set, the better the algorithm can model the behavior of the system. This result also demonstrates one of the drawbacks of using this system. Not all missions are the same, and telemetry that corresponds to one mission may not be what is experienced by another. Therefore training data must be carefully selected and used only for missions and features that should experience similar behavior. In our case, all but this Atmel sensor conformed to the training data, which is still an overall good result. In this case, we would include in that training set, this new data that we received from the ISX power model so we can adapt the system to produce better results. After including this new data in the training set, the model performed far better and we saw much lower deviation scores which could just be attributed to noise.

6.3.3 Fault injection

For a final test, a simulated error was injected into the system running on ISX to see if the monitoring module could pick up the anomaly. For this test, the temperature module was used and a new feature that measured the temperature of the negative Z side panel was added to the feature set. This new module was trained and placed in the file hierarchy. Since the side panels were not attached, a default value of zero would be returned any time that sensor was queried thereby simulating a “fault”. Since the training data contained actual values for this sensor, it should produce an anomaly by the system.

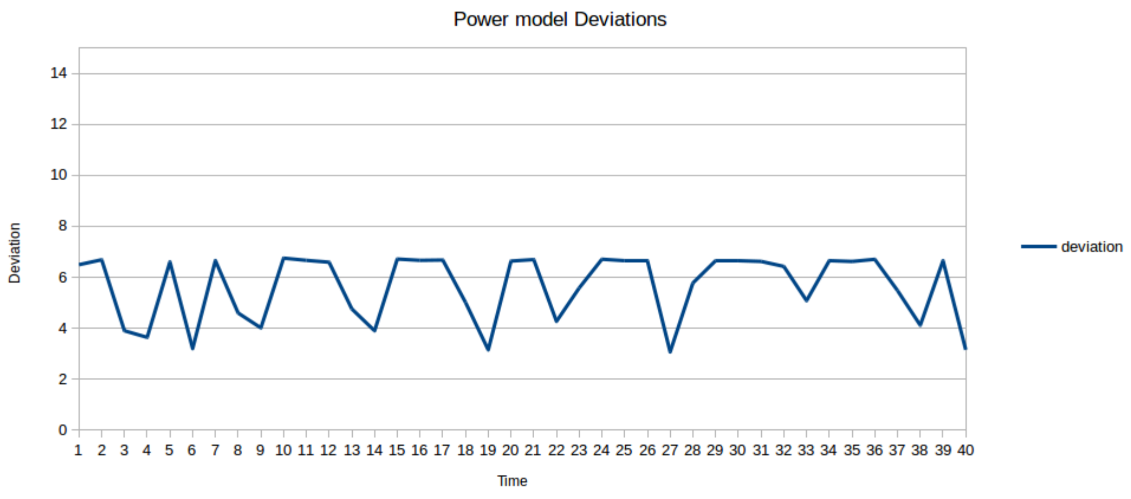


Figure 6.9: Deviation scores produced by run of power model on ISX

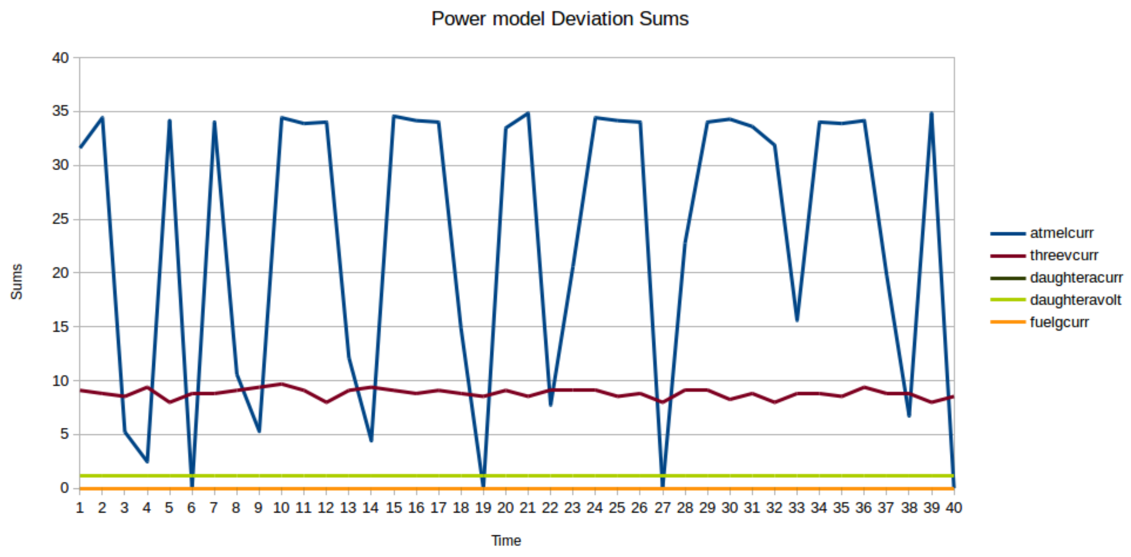


Figure 6.10: Deviation Sums for top five sensors that contributed the most error

The individual deviation sums of each feature in this run are shown in Figure 6.11. The deviation sums seen in the first half of this chart are the results from before this fault injection. As can be seen, the behavior was nominal and resulted in no deviations. The fault injection occurs when you see the steep increase in the deviation value on the *nz_panel* sensor which should reside on the non-existent side panel. This results shows that the system was able successfully detect the anomaly and identify which feature was causing the issue.

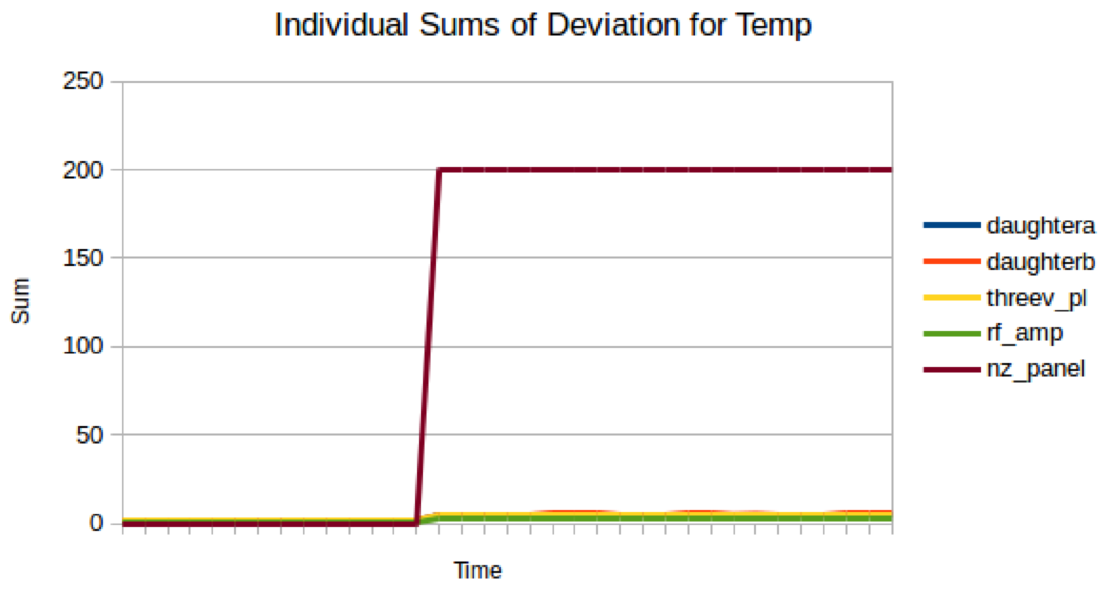


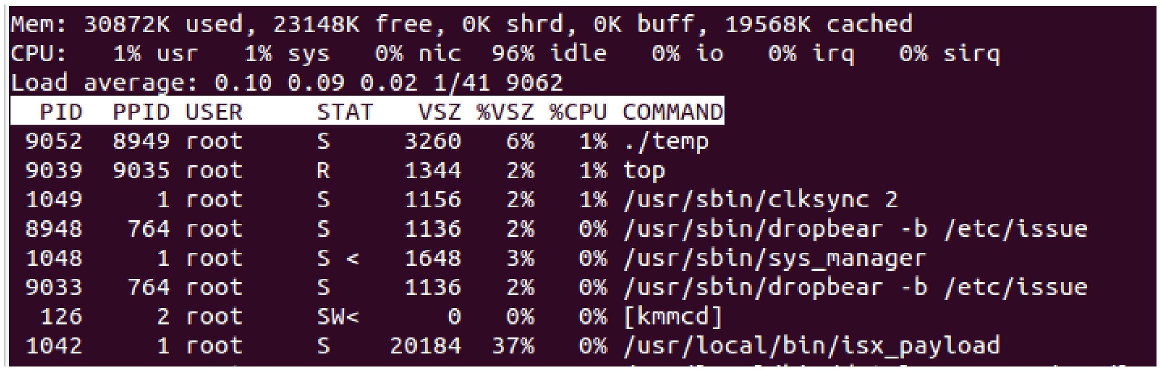
Figure 6.11: The feature deviation sums after fault injection

6.4 Resources and efficiency

The new monitoring library's drain on system resources was also important to test. Too much resource usage in terms of memory and processing is indicative of a slow and inefficient system. We want the monitoring to be done fast so it is important that as new telemetry comes in, the system is able to immediately process and make a prediction without using too much system memory.

The monitoring library has to make UDP calls to different processes to get the feature values, and also has to go through an array of clusters to locate the one with the minimal distance. This introduces a certain level of processing time as all this information is gathered. We tested the system with two modules that together contained 1600 clusters with 23 features. This is representative of a relatively large monitoring load. Each time the anomaly event fired, the resulting deviation value would be available in less the 70 milliseconds. This speed is well within the range that we were aiming for. Since the check anomaly event is scheduled to run every 30 seconds, we should see no problems at all for getting a result on time. This fast speed also means that there is not much processing being done and we have an efficient algorithm that is using minimal computing resources.

Using Unix's built in **top** task management utility, the max CPU usage was only 3% when the check anomaly event was processing. Most of the time it would stay at around 1%. These results that we see with the speed and processing required, fit with our requirement of making the system fast and efficient. The output of the **top** utility can be seen in Figure 6.12, where ./temp is the process running the monitoring.



```
Mem: 30872K used, 23148K free, 0K shrd, 0K buff, 19568K cached
CPU:   1% usr   1% sys   0% nic  96% idle   0% io   0% irq   0% sirq
Load average: 0.10 0.09 0.02 1/41 9062
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
 9052  8949 root     S     3260   6%   1% ./temp
 9039  9035 root     R     1344   2%   1% top
 1049     1 root     S     1156   2%   1% /usr/sbin/clksync 2
 8948   764 root     S     1136   2%   0% /usr/sbin/dropbear -b /etc/issue
 1048     1 root     S <   1648   3%   0% /usr/sbin/sys_manager
 9033   764 root     S     1136   2%   0% /usr/sbin/dropbear -b /etc/issue
  126     2 root     SW<      0   0%   0% [kmmcd]
 1042     1 root     S    20184  37%   0% /usr/local/bin/isx_payload
```

Figure 6.12: Resource usage by the monitoring library

The memory usage of the monitoring library was also examined using the **top** utility. With the same test setup mentioned above, the library was using about 3.2 MB or 6% of the total amount of virtual memory in the system. This is with a relatively large amount of clusters allocated to the memory space. This is a reasonable amount of memory that the monitoring library is using. Most of this memory is used by the data structure defining each cluster. Some improvements can be made on this percentage by using 4 byte floats to store the feature values instead of 8 byte doubles. Also some optimization's can be made in the code to further reduce this memory usage. However as of now, the memory used by the monitoring system can be deemed acceptable. As the number of modules and cluster's increase, some improvements to the memory usage may have to be made.

Overall, the resource usage and efficiency of the system meet the goal of our requirements.

Chapter 7

CONCLUSION

The application of the Inductive Monitoring System for system health monitoring of CubeSat satellites has great potential. Such a system can be accurately and efficiently used to monitor for anomalies in a size and resource restricted CubeSat satellite. Archived telemetry can be used by data driven health monitoring techniques such as IMS to characterize models of nominal system behavior from the data itself instead of having to rely on more traditional parameter checking methods or more complicated model based techniques. As the amount of archived telemetry and data increases the more missions that are flown, these models can be updated to provide better and more accurate results that generate a more comprehensive model of the system's behavior.

IMS was used in this thesis as a tool to analyze and detect errors in archived flight telemetry, and as a system that was integrated into PolySat's flight software to provide real-time monitoring of anomalies. The system was designed to meet the goals of efficiency, low resource consumption, genericity, and adaptability that are important precursors for use in CubeSat satellites. The end product is a system that generates the training models on the ground and uploads them to the satellite which runs a monitoring library to produce deviation scores and anomaly reports. This monitoring system will be used in all future missions for anomaly detection.

The results of our tests indicate that the system performed very well in finding errors in archived flight telemetry and real time monitoring given good training models. The training data used must be comprehensive to include all possible system behavior. Since this requirement is somewhat impractical, the ability for the system to update itself given new flight telemetry allows for a more accurate representation of the system as the mission progresses. The importance of well correlated feature sets for monitoring was also examined so the system can perform at its best and have the least amount of false positives. The resource usage and efficiency of the integrated system was well within the limits of the resource constrained CubeSats, making such a system completely practical for use. The success of this system makes it perfect for use as a autonomous tool for system health monitoring in all future PolySat missions.

7.1 Future Work

There were quite a few things in this thesis that weren't fully explored. They are a perfect opportunity for future work by any new PolySat lab member.

7.1.1 Multiple models for different modes of operation

The current system characterizes only one model per module for the entire system behavior of the spacecraft. However, throughout its mission, a CubeSat can change its mode of operation many times which can cause completely different system behavior. Examples of these different modes of operations include when the spacecraft is idle, or when it is running an experiment. For these cases, system behavior changes significantly in a way that may not be well defined in one big model.

In these cases it is best to create a separate model for each mode of operation and switch between models as these modes change. This leads to much more accurate and reliable results because the system adapts itself to use a model more consistent with the current system behavior. This ability was not possible to currently implement in the system because the archived IPEX data was limited and did not contain clearly defined regions of different modes. However as future missions progress and more telemetry is gathered, these different modes can be characterized and used in the monitoring library. There currently exists a placeholder in the monitoring library that allows for this functionality to be added so any future interested lab member can work on this.

7.1.2 Dynamic cluster expansion

IMS can be extended to add the ability to dynamically expand its clusters while the monitoring phase is running. This ability allows for the capture of additional system behavior that is close enough to existing models that was not captured in the training data. The clusters that exist can be expanded to include new nominal data vectors that are within a certain range of existing clusters. In a way, it allows for a certain extent of training to occur while the mission is in progress. This ability can be extremely useful in cases where the archived telemetry is limited and some system behavior needs to be learned as the mission is in progress. This extension can be added to the monitoring library to make the system more robust to different environmental conditions that are seen from mission to mission.

7.1.3 Analysis of other anomaly detection algorithms

Other anomaly detection algorithms such as ORCA, one-class SVMs, and Virtual sensors can be explored and implemented as an additional tool that can be used alongside IMS to provide anomaly detection. The results and detection capabilities of these algorithms can be compared and contrasted to see which algorithm provides the best results and to see if it is feasible to run multiple systems efficiently on the CubeSat. Such an exploration could improve upon the accuracy of the current system and provide a great opportunity for research and learning of other data-driven machine learning algorithms.

BIBLIOGRAPHY

[1] The cubesat program,. `http://www.cubesat.org/about/`. Accessed: 2016-11-12.

[2] One-class svm with non-linear kernel (rbf).

[3] Polysat,. `http://polysat.calpoly.edu/`. Accessed: 2016-11-20.

[4] Supervised and unsupervised machine learning algorithms, Sept. 2016.

[5] E. Alpaydn. *Introduction to Machine Learning*. MIT Press, 2 edition, 2010.

[6] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 29–38. ACM, 2003.

[7] R. H. Chen, H. K. Ng, J. L. Speyer, L. S. Guntur, and R. Carpenter. Health monitoring of a satellite system. *Journal of guidance, control, and dynamics*, 29(3):593–605, 2006.

[8] E. Howell. Columbia disaster: What happened, what nasa learned, Feb. 2013.

[9] D. Iverson. Inductive monitoring system constructed from nominal system data and its use in real-time system monitoring, June 3 2008. US Patent 7,383,238.

[10] D. Iverson, R. Martin, M. Schwabacher, L. Spirkovska, W. Taylor, R. Mackey, and J. Castle. General purpose data-driven system monitoring for space operations, 2009 aiaa infotech@ aerospace conference. *Seattle, WA, Apr*, 2009.

[11] D. L. Iverson. Inductive system health monitoring. 2004.

[12] D. L. Iverson. Inductive system health monitoring with statistical metrics. 2005.

[13] D. L. Iverson. System health monitoring for space mission operations. In *Aerospace Conference, 2008 IEEE*, pages 1–8. IEEE, 2008.

[14] S. Lee, A. Hutputanasin, A. Toorian, W. Lan, and R. Munakata. Cubesat design specification, Aug. 2009.

[15] G. Manyak. Fault tolerant and flexible cubesat software architecture. *Cal Poly Digitial Commons*, 2011.

[16] G. Manyak and J. M. Bellardo. Polysat's next generation avionics design. In *Space Mission Challenges for Information Technology (SMC-IT), 2011 IEEE Fourth International Conference on*, pages 69–76. IEEE, 2011.

[17] R. Martin, M. Schwabacher, N. Oza, and A. Srivastava. Comparison of unsupervised anomaly detection methods for systems health management using space shuttle. In *Main Engine Data, Proceedings of the Joint Army Navy NASA Air Force Conference on Propulsion, 2007*. Citeseer, 2007.

[18] B. L. Matthews, A. N. Srivastava, D. Iverson, B. Beil, and B. Lane. Space shuttle main propulsion system anomaly detection: A case study. *IEEE Aerospace and Electronic Systems Magazine*, 26(9):4–13, 2011.

[19] J. Puig-Suari, C. Turner, and R. Twiggs. Cubesat: the development and launch support infrastructure for eighteen different satellite customers on one launch. 2001.

[20] A. Smola. *Introduction to Machine Learning*. University of Cambridge, 2008.

[21] D. M. Tax and R. P. Duin. Support vector domain description. *Pattern recognition letters*, 20(11):1191–1199, 1999.

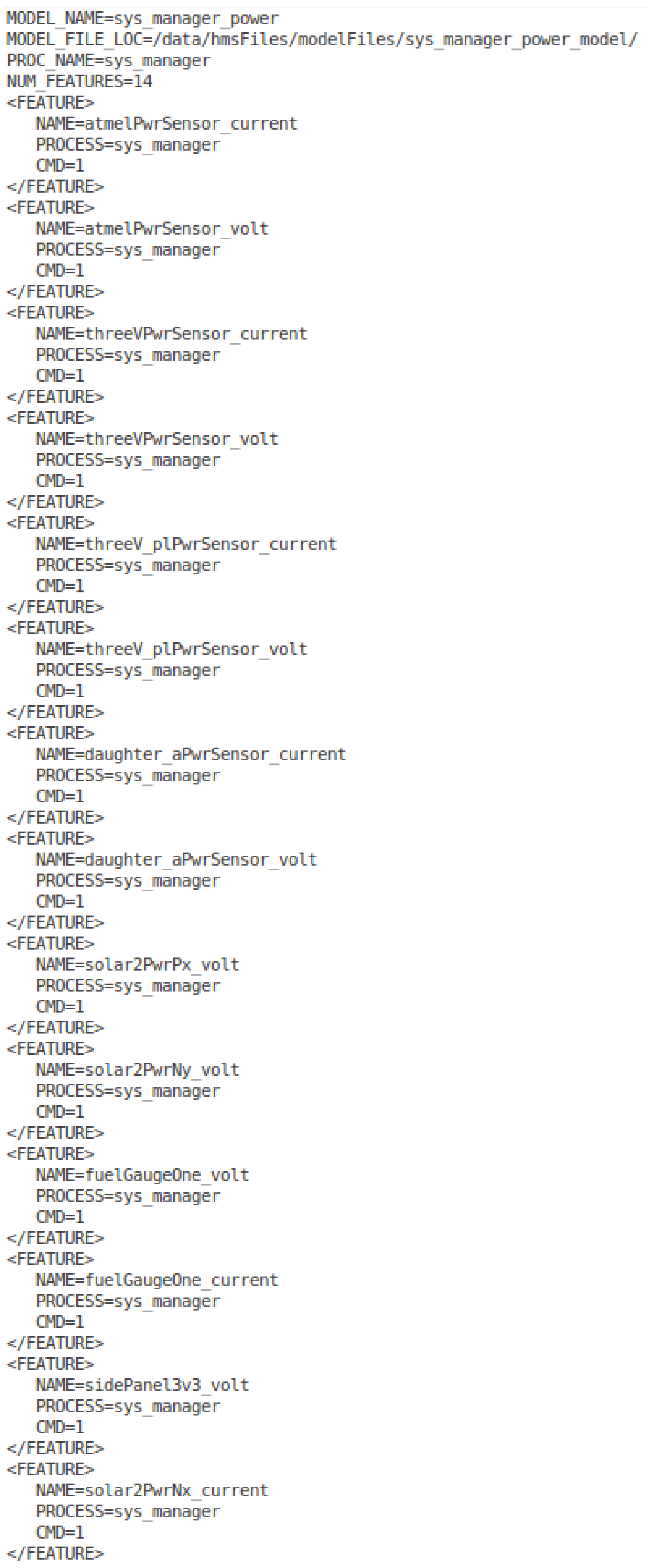
[22] I. H. Witten and I. H. Witten. *Data mining: practical machine learning tools and techniques.* Elsevier, 2017.

[23] C. Zaiontz. Basic concepts of correlation, Feb. 2013.

Appendix

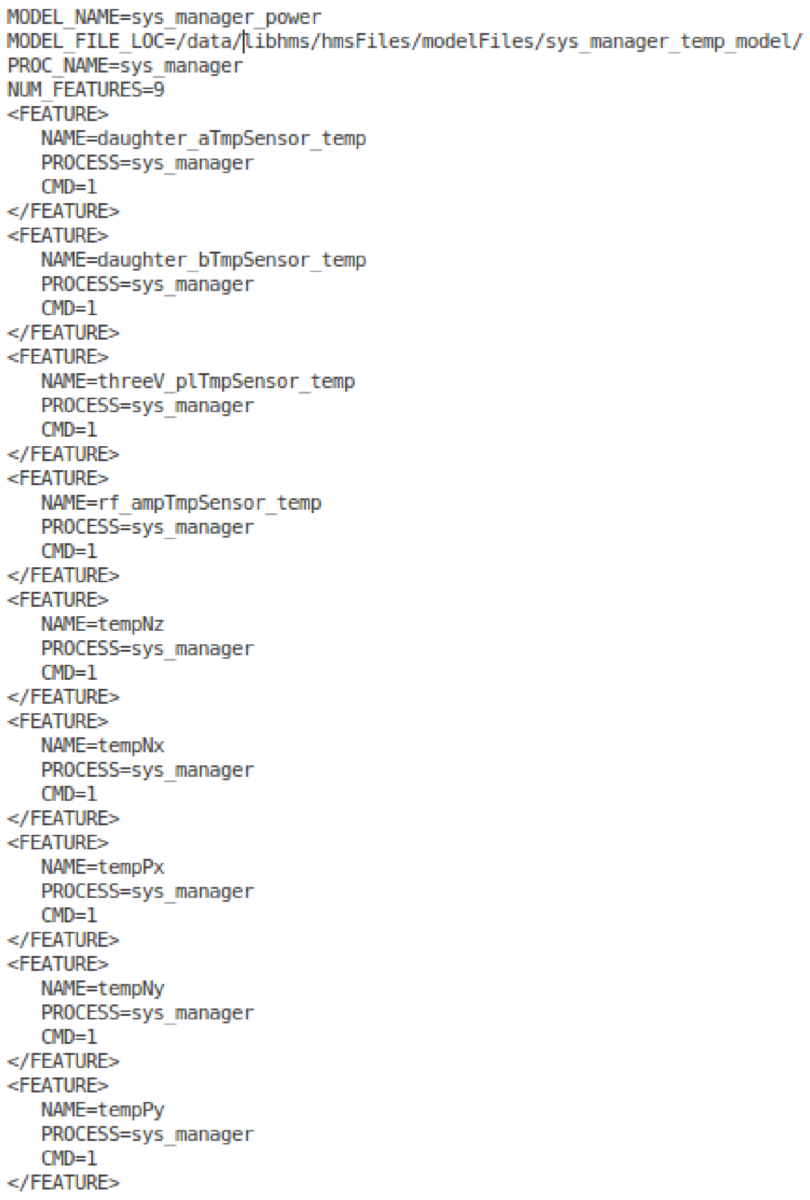
CONFIGURATION FILES FOR MODELS

This appendix contains the configuration files that were used in this thesis to select features for training and monitoring. There were two models created, one for the set of power sensors and the other for the set of temperature sensors. Each file contains the name of the model to which this configuration file corresponds. It also contains the file path of the corresponding cluster file and gives the name of the process to which this model belongs. The file also specifies the total number of features in the model and for each feature gives its sensor name, the name of the process that owns the it, and the number of the command that needs to be sent to obtain its value. Testing used the full set or a subset of these features listed.



```
MODEL_NAME=sys_manager_power
MODEL_FILE_LOC=/data/hmsFiles/modelFiles/sys_manager_power_model/
PROC_NAME=sys_manager
NUM_FEATURES=14
<FEATURE>
   NAME=atmelPwrSensor_current
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=atmelPwrSensor_volt
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=threeVPwrSensor_current
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=threeVPwrSensor_volt
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=threeV_plPwrSensor_current
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=threeV_plPwrSensor_volt
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=daughter_aPwrSensor_current
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=daughter_aPwrSensor_volt
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=solar2PwrPx_volt
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=solar2PwrNy_volt
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=fuelGaugeOne_volt
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=fuelGaugeOne_current
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=sidePanel3v3_volt
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=solar2PwrNx_current
   PROCESS=sys_manager
   CMD=1
</FEATURE>
```

Figure A.1: Configuration file for the full Power Model



```
MODEL_NAME=sys_manager_power
MODEL_FILE_LOC=/data/libhms/hmsFiles/modelFiles/sys_manager_temp_model/
PROC_NAME=sys_manager
NUM_FEATURES=9
<FEATURE>
   NAME=daughter_aTmpSensor_temp
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=daughter_bTmpSensor_temp
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=threeV_plTmpSensor_temp
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=rf_ampTmpSensor_temp
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=tempNz
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=tempNx
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=tempPx
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=tempNy
   PROCESS=sys_manager
   CMD=1
</FEATURE>
<FEATURE>
   NAME=tempPy
   PROCESS=sys_manager
   CMD=1
</FEATURE>
```

Figure A.2: Configuration file for the full Temperature Model