ANOMALY DETECTION USING MACHINE LEARNING METHODS

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Electrical Engineering

By

David A. Dozal

2017

SIGNATURE PAGE

THESIS:	ANOMALY DETECTION USING MACHINE LEARNING METHODS
AUTHOR:	David A. Dozal
DATE SUBMITTED:	Winter 2017
	Electrical and Computer Engineering Department

Dr. Zekeriya Aliyazicioglu _____
Thesis Committee Chair
Electrical and Computer Engineering
Department

Dr. Rajan Chandra _____
Electrical and Computer Engineering
Department

Dr. Tim Lin _____
Electrical and Computer Engineering
Department

ACKNOWLEDGEMENTS

First, I would like to thank my research advisers, Dr. Zekeriya Aliyazicioglu and Dr. Rajan Chandra for their help and guidance throughout this research project. I have learned so much from them on conducting research and how to be a trained professional. I am very grateful to Dr. Tim Lin for teaching Natural Language Processing and Network Security courses, which were beneficial towards this thesis. I would like to thank my research colleague John Wu who worked with me on anomaly detection research and gave new ideas towards this research. I would also like to express my sincere appreciation to Northrop Grumman for funding the anomaly detection research that I worked on for the past year and providing me the opportunity to explore anomaly detection systems. Finally, I would like to thank my mother Sylvia who has been very supportive throughout my college education.

ABSTRACT

Single event upsets can pose a serious threat to satellite systems in space. In space ionizing radiation strikes can generate unwanted effects in the memory cell of semiconductor devices, which can cause anomalies to occur. These types of anomalies are a major problem in the reliability and longevity of satellite systems. The goal of this research is to find a machine learning algorithm that can accurately detect anomalies. The machine learning techniques that were investigated for anomaly detection are Kernel Principal Component Analysis (KPCA), K-Means Clustering, and Recurrent Neural Networks (RNN).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Anomaly detection systems are very important to detect and identify unwanted abnormal behaviors. These types of systems are being developed to detect anomalies in satellite systems, which are caused by single event upsets (SEUs). Single event upsets are a result of ionizing radiation strikes that can generate unwanted effects in the latch state or memory cell of semiconductor devices. This type of radiation occurs in satellite systems in space, which can pose a serious threat to computer reliability and longevity.

Northrop Grumman provided satellite network datasets, which included single event upsets. The two types of datasets that were provided for this research were nominal data and off nominal data. The nominal data have no anomalies while the off nominal data have anomalies. In the two datasets, there were 55 columns of data that represented the whole satellite network.

The goal of this research is to find a machine learning algorithm that can accurately detect satellite network anomalies. The machine learning algorithm needs to have a low false alarm rate to make it practical to be deployed in real satellite network. The accuracy, performance, and reliability of these algorithms need to be evaluated with various metrics methods. The most important measurements to evaluate the algorithms with is the sensitivity and specificity, misclassification rate, confusion matrix, precision and recall, F-measure, and receiver operating characteristic (ROC) curves. By using each of these evaluation algorithms on the different machine learning methods the best algorithm for anomaly detection can be determine.

Machine learning techniques can be used to help make intelligent decisions or predictions when anomalous behaviors occur on the network. The challenge of detecting

different types of anomalies can be handle with many different types of machine learning techniques. The machine learning techniques that were explored for anomaly detection are Kernel Principal Component Analysis (KPCA), K-Means Clustering, and Recurrent Neural Networks (RNN).

Previous research on anomaly detection with machine learning was expanded in this thesis. The research provided by Heiko Hoffmann, author of *Kernel PCA for Novelty Detection*, was used to test simple anomaly detection examples on small data sets. In this thesis KPCA was used on large data sets that came from satellite systems. Ted Dunning and Ellen Friedman, authors of *Practical Machine Learning*, research explored anomaly detection by using windowing and clustering with a percentile threshold to flag anomalies. One of the issues in this research was that there were too many false alarms. This was solved by using a sliding window method to flag anomalies if there are a specific number of anomalies above the threshold of the window. This method reduced the number of false alarms. Past research of Recurrent Neural Networks mainly focus on Natural Language Processing. In this thesis RNN focuses on detecting anomalies by training the RNN to reconstruct the original signal with no anomalies. After training the RNN model it can then test the signal with anomalies. If the RNN can't reconstruct the signal then there must be anomalies where the reconstruction error is the highest.

The remainder of this thesis is organized as follows. Chapter 2 focuses on introducing Principal Component Analysis. This chapter reviews the algorithm for Principal Component Analysis and covers the different applications that Principal Component Analysis can be used for. In Chapter 3, Kernel Principal Component Analysis algorithm is explored for anomaly detection. This chapter discusses how to use the kernel

trick to change Principal Component Analysis into Kernel Principal Component Analysis to handle nonlinear functions and using the reconstruction error to detect anomalies.

Chapter 4 covers the windowing method used to create a nominal data waveform dictionary. In this chapter, K-Means Clustering was used to reduce the number of waveform dictionary component shapes and create a new cluster waveform dictionary. This chapter explains into detail on how to use the nominal data cluster waveform dictionary for anomaly detection. Chapter 5 presents the idea of using Recurrent Neural Networks for anomaly detection. It covers the Recurrent Neural Network algorithm, Long Short Term Memory, and the problems that come from training Recurrent Neural Networks. The thesis concludes in Chapter 6 with a summary and directions for future research.

CHAPTER 2: PRINCIPAL COMPONENT ANALYSIS

The fundamental idea of Principal Component Analysis (PCA) is to reduce the dimensionality of a data set consisting of many correlated variables, while retaining as much variance as possible in the data set. This is achieved by transforming to a new set of variables, the Principal Components (PCs), which are uncorrelated, and are ordered so that the first few retain most of the variation present in all the original variables [1]. The following are important properties for PCA:

1. PCA tries to find directions that maximizes the variance of the data.
2. PCA finds directions that are mutually orthogonal. It is a global algorithm since it has a global constraint.
3. PCA gives the best reconstruction error.
4. Minimizes the least square error when reducing the dimensionality of a data set.
5. If the eigenvalue of any dimension is equal to zero, then it provides no additional information in the original space. This value can be ignored without affecting your reconstruction error.

2.1 Principal Component Analysis Algorithm

To perform Principal Component Analysis, the data first needs to be preprocess by normalizing the data to have a comparable range of values. Given a training set of $x_1, x_2, \dots, x_n$ where n is the number of features. To normalize the training set, each feature x_n needs to be normalize by the following equation,

$$\frac{x_n - \mu_n}{\sigma_n}, \tag{2.1}$$

where x_n is the data point, μ_n is the mean, and σ_n is the standard deviation. Normalizing the data is very useful when multiple features are measured in different scales.

After normalizing the training set data, the next step is to reduce the data from n-dimensions to k-dimensions. The variance-covariance matrix first needs to be computed with the following equation,

$$\boldsymbol{K} = \frac{1}{m}\boldsymbol{X}^T\boldsymbol{X}, \tag{2.2}$$

where m is the number of samples and $\boldsymbol{X}$ is the training set data matrix.

Singular value decomposition can then be used to calculate the singular vectors and singular values of the variance-covariance matrix $\boldsymbol{K}$,

$$\boldsymbol{K} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T, \tag{2.3}$$

where $\boldsymbol{U}$ is the singular vector matrix of $\boldsymbol{K}\boldsymbol{K}^{\boldsymbol{T}}$, $\boldsymbol{S}$ is the singular value matrix, and $\boldsymbol{V}$ is the singular vector matrix of $\boldsymbol{K}^{\boldsymbol{T}}\boldsymbol{K}$. From the singular vector matrix $\boldsymbol{U}$,

$$\boldsymbol{U} = [\bar{u}_1 \quad \bar{u}_2 \quad \cdots \quad \bar{u}_n] \in \mathbb{R}^{nxn}, \tag{2.4}$$

we can get the singular vectors of $\boldsymbol{K}\boldsymbol{K}^{\boldsymbol{T}}$ by using the first k columns of the matrix $\boldsymbol{U}$. This can reduce the training set data from n-dimensions to k-dimensions to form a new reduced matrix $\boldsymbol{Z}$,

$$\boldsymbol{Z} = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} \begin{bmatrix} u_{11} & \cdots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{n1} & \cdots & u_{nk} \end{bmatrix} = \boldsymbol{X}\boldsymbol{U}_{reduced}, \tag{2.5}$$

where n is the number of features, m is the number of samples, k is the number of principal components, and $\boldsymbol{X} \in \mathbb{R}^{m\times n} \Longrightarrow \boldsymbol{Z} \in \mathbb{R}^{m\times k}$.

To reconstruct the transformed data back to its original dimension, $\boldsymbol{X}_{approx}$ needs to be calculated for each data point with the following equation,

$$\boldsymbol{X}_{approx} = \begin{bmatrix} z_{11} & \cdots & z_{1k} \\ \vdots & \ddots & \vdots \\ z_{m1} & \cdots & z_{mk} \end{bmatrix} \begin{bmatrix} u_{11} & \cdots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{n1} & \cdots & u_{nk} \end{bmatrix}^T = \boldsymbol{Z}\boldsymbol{U}_{reduced}^T, \tag{2.6}$$

where $\boldsymbol{Z} \in \mathbb{R}^{m \times k} \Longrightarrow \boldsymbol{X} \in \mathbb{R}^{m \times n}$. This can allow the training set data in k dimensions to be reconstructed to n dimensions.

Choosing the correct number of principal components is very important to preserve as much variance possible while reducing the dimension size of the data. To choose the correct number of k principal components to retain 99% of variance the following must apply,

$$\frac{\text{Average Square Projection Error}}{\text{Total Variation}} = \frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x(i) - x_{approx}(i)\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x(i)\|^2} \leq 0.01, \tag{2.7}$$

where x is the actual data and x_{approx} is the reconstructed data.

A simpler way of finding the correct k principal components is to use the singular value matrix from singular value decomposition. From the singular value matrix $\boldsymbol{S}$, we can get the following singular values,

$$\boldsymbol{S} = \begin{bmatrix} S_{11} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & S_{nn} \end{bmatrix}. \tag{2.8}$$

The best way to find the best number of principal components to use is to perform PCA with $k = 1, 2, \ldots, n$ until the smallest value of k satisfy the amount of variance that is needed to be retained. This can be accomplished with the following equation,

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x(i) - x_{approx}(i)\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x(i)\|^2} = 1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \leq 1 - v \Longrightarrow \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \geq v, \tag{2.9}$$

where S is the singular values, n is the number of features, k is the number of principal components, and v is the variance that is wanted to be retained. By just observing the singular values matrix the process of finding the correct number of principal components is greatly simplified.

2.2 Applications of PCA

Principal Component Analysis (PCA) can be used for many different types of applications. One of the most popular applications that PCA is used for is data compression. Data compression reduces the memory space needed to store any type of data. A great example for this is to compress the data of an image. The two pictures in figure 2.1 are images of my professors Zeki and Chandra. The first step before compressing these images is to add white Gaussian noise to them to see how well you can reconstruct the image if noise is applied. In figure 2.2 the white Gaussian noise is added to these two images. After adding noise to the two images PCA can then be performed.

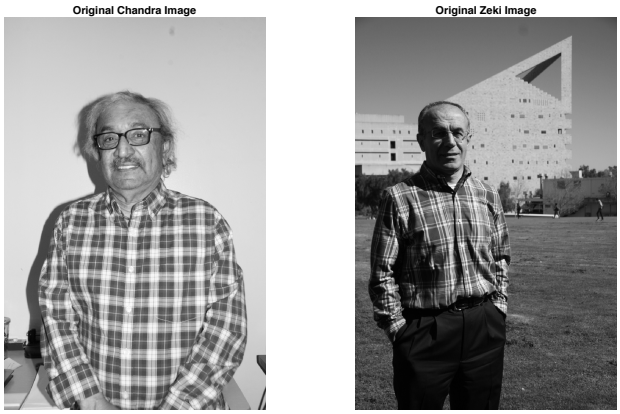

Figure 2.1 Original Images of Zeki (Right) and Chandra (Left)

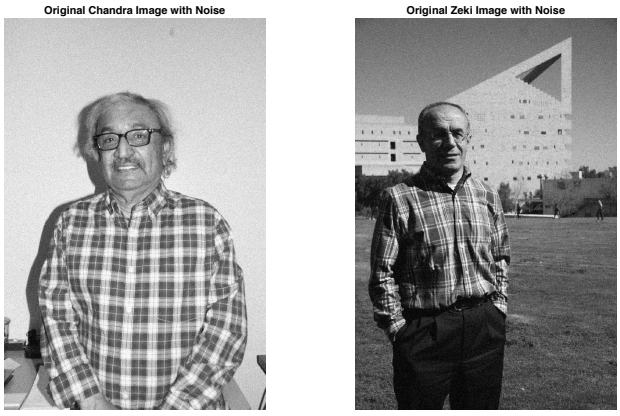

Figure 2.2 Original Images of Zeki (Right) and Chandra (Left) with Noise

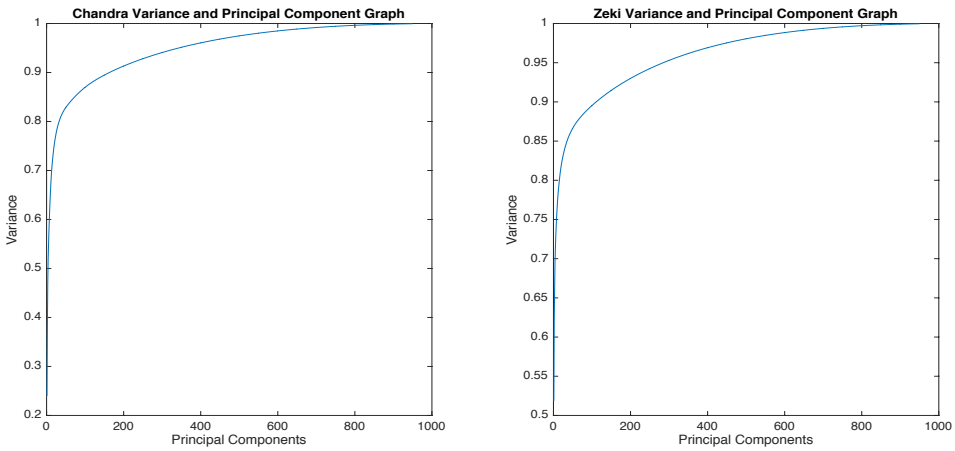

Figure 2.3 Zeki (Right) and Chandra (Left) Variance and Principal Component Graphs

Choosing the correct number of principal components is very important when preforming PCA. The amount of variance retained per principal component can be shown in figure 2.3 graphs. This is useful because you can determine the exact number of principal components needed for retaining each percentage of variance. If too few principal components are used, then the quality of the image is greatly affected since too little of the variance was retained.

In figure 2.4 the reconstruction of the images is very poor since only 80% of the variance was retained. By only retaining 80% of the variance the total number of principal components was reduce from 1000 to 16 for Zeki and 33 for Chandra images. The data size of the images was greatly decreased but at the cost of not retaining the image quality of the original picture. To get a near perfect image reconstruction the amount of variance that needs to be retain is around 99%. We can see in figure 2.5 that the reconstruction of the images is almost perfect when 99% variance retained. By retaining just 99% of the variance the total number of principal components was reduce from 1000 to 625 for Zeki and 670 for Chandra images. PCA is a perfect tool to reduce and compress data in images.

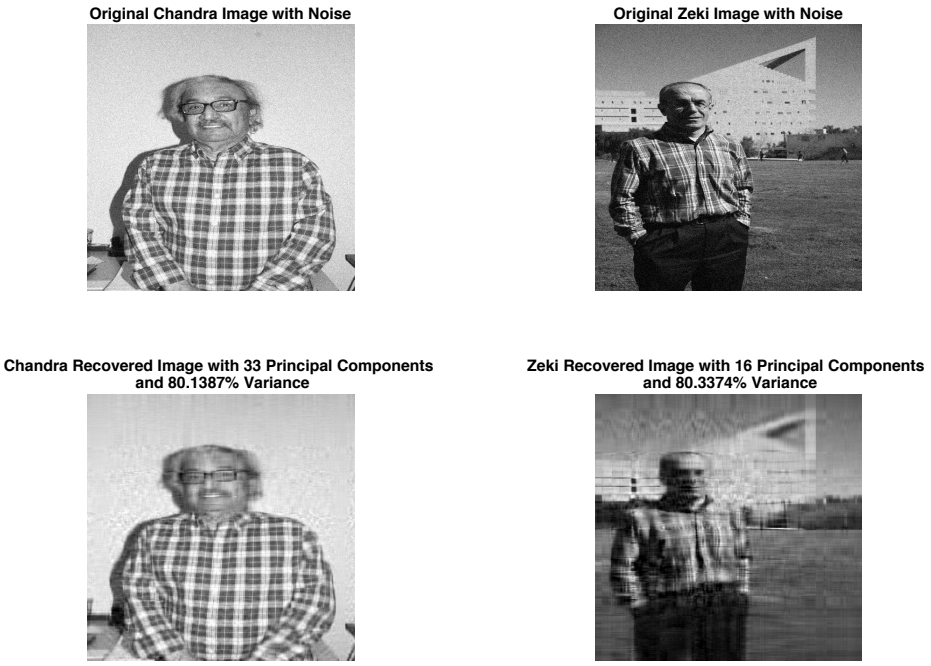

Figure 2.4 Zeki (Right) and Chandra (Left) Recovered Image with 80% Variance

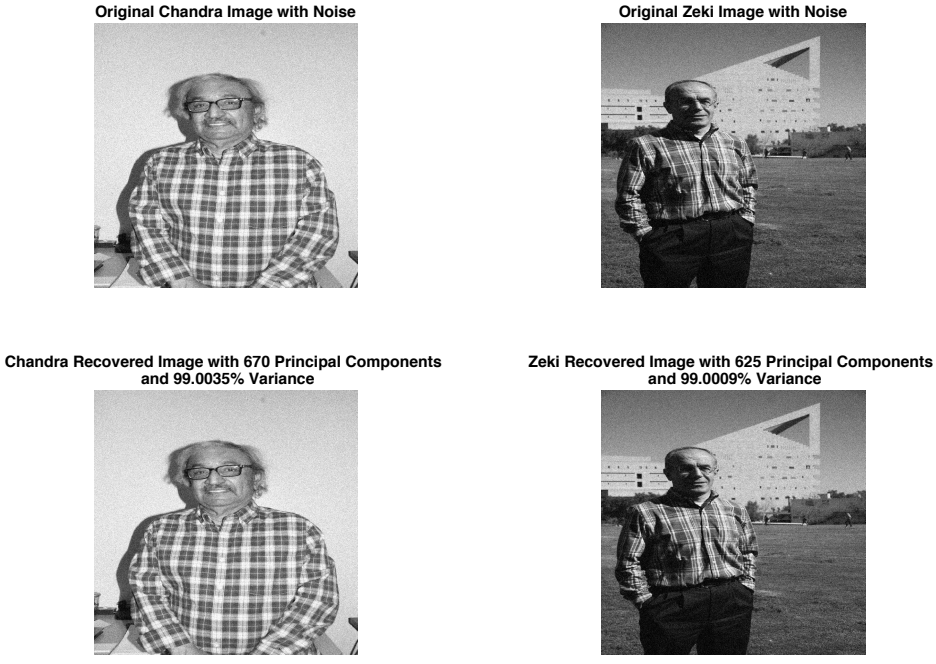

Figure 2.5 Zeki (Right) and Chandra (Left) Recovered Image with 99% Variance

PCA is also popular for facial recognition. This can be achieved by finding all the eigenfaces in the face dataset. By finding the eigenfaces you can compare each face with each other to see how similar they are to each other for facial recognition. The first step for facial recognition is to create a face dataset. In figure 2.6 the faces of the professors from Cal Poly Pomona were used to create a face dataset. After creating a face dataset, PCA can then be performed. The principal components of each professor's face can be displayed in figure 2.7. These face images are known as the eigenfaces.

Figure 2.6 Cal Poly Pomona Face Dataset

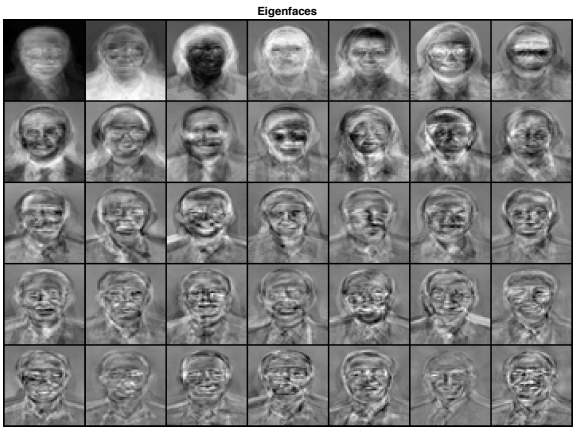

Figure 2.7 Eigenfaces of Cal Poly Pomona Face Dataset

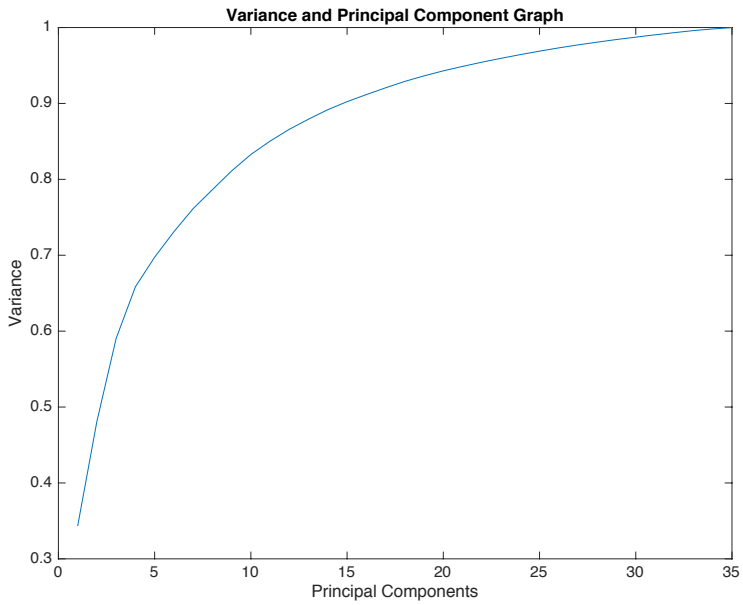

Figure 2.8 Cal Poly Pomona Face Dataset Variance and Principal Component Graph

After finding the eigenfaces of Cal Poly Pomona face dataset the original face dataset can be recovered with PCA. Depending on the amount of variance needed to be retain the quality of the recovered face dataset will be affected. Figure 2.8 shows the variance and principal component graph of the Cal Poly Pomona face dataset. When the principal components are increased, the variance also increases. This feature of PCA is very important when deciding the number of principal components.

In figure 2.9 the face dataset reconstruction was very poor since only 81% of the variance was retained. The faces of the recovered images are unclear and blurry. The total number of principal components was reduced from 36 to 9 for 81% variance. To get a higher quality image for every face the number of principal components needs to be increase. We can see in figure 2.10 that the variance was increase to 99% by using 31 principal components. The faces images that were recovered here were very clear and almost identical to the original faces. All the different types of applications PCA can be used for makes PCA a very powerful and useful algorithm.

Figure 2.9 Cal Poly Pomona Face Dataset Reconstruction with 81% Variance

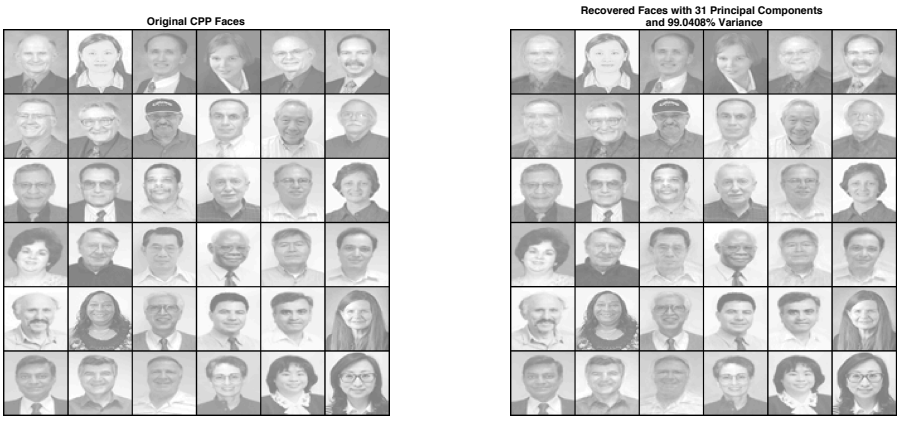

Figure 2.10 Cal Poly Pomona Face Dataset Reconstruction with 99% Variance

CHAPTER 3: KERNEL PRINCIPAL COMPONENT ANALYSIS

Kernel Principal Component Analysis (KPCA) is a non-linear extension of PCA that can work very well with non-linear data. One of the main problems with using standard PCA is that it doesn't perform very well with non-linear data sets. To solve this problem KPCA takes advantage of the kernel trick to replace all the scalar products with a kernel function. This allows the data to be mapped into a higher-dimensional space called feature space, while the algorithm still operates in the original space. In this new higher-dimensional space standard PCA will then be computed.

3.1 KPCA Algorithm

According to Bernhard Schölkopf and Alexander J. Smola, authors of *Learning with Kernels*, to perform KPCA the data must be projected to a high-dimension feature space $\boldsymbol{\mathcal{H}}$,

$$\begin{aligned} \Phi: \mathcal{X} &\longrightarrow \boldsymbol{\mathcal{H}}, \\ x_n &\longrightarrow \Phi(x_n), \end{aligned} \tag{3.1}$$

where $x_n \in \mathbb{R}^d \longrightarrow \Phi(x_n) \in \mathbb{R}^D$, and $d \ll D$. In the feature space $\boldsymbol{\mathcal{H}}$ the $M \times M$ covariance matrix takes the form,

$$\boldsymbol{C} = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{\Phi}(x_n) \boldsymbol{\Phi}(x_n)^T , \tag{3.2}$$

where N is the number of samples and M is the number of features. Next the principal components need to be computed by solving the eigenvalue problem in feature space $\boldsymbol{\mathcal{H}}$,

$$\boldsymbol{\lambda}_m \boldsymbol{V}_m = \boldsymbol{C} \boldsymbol{V}_m, \tag{3.3}$$

where $m = 1, 2, \ldots, M$, the eigenvalues $\boldsymbol{\lambda} \geq 0$, and the nonzero eigenvectors $\boldsymbol{V} \in \boldsymbol{\mathcal{H}}$.

The eigenvector V_m of the covariance matrix in feature space $\mathcal{H}$ is a linear combination of points $\boldsymbol{\Phi}(x_n)$,

$$\boldsymbol{V}_m = \sum_{n=1}^{N} \boldsymbol{\alpha}_{mn}\, \boldsymbol{\Phi}(x_n). \tag{3.4}$$

The kernel function needs to be added to the eigenvalue problem by multiplying both sides of equation $\boldsymbol{\lambda}_m \boldsymbol{V_m} = \boldsymbol{C V_m}$ by $\boldsymbol{\Phi}(x_k)^T$,

$$\boldsymbol{\lambda}_m \langle \boldsymbol{\Phi}(x_k), \boldsymbol{V_m} \rangle = \langle \boldsymbol{\Phi}(x_k), \boldsymbol{C V_m} \rangle, \tag{3.5}$$

$$\boldsymbol{\lambda}_m \sum_{n=1}^{N} \boldsymbol{\alpha}_{mn} \langle \boldsymbol{\Phi}(x_k), \boldsymbol{\Phi}(x_n) \rangle = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{\alpha}_{mn} \left\langle \boldsymbol{\Phi}(x_k), \sum_{j=1}^{N} \boldsymbol{\Phi}\big(x_j\big) \big\langle \boldsymbol{\Phi}\big(x_j\big), \boldsymbol{\Phi}(x_n) \big\rangle \right\rangle, \tag{3.6}$$

where $k = 1, 2, \ldots, N$. In terms of $N \times N$ Gram matrix $\boldsymbol{K}_{ij} = \big\langle \boldsymbol{\Phi}(x_i), \boldsymbol{\Phi}\big(x_j\big) \big\rangle$,

$$N \boldsymbol{\lambda K \alpha} = \boldsymbol{K}^2 \boldsymbol{\alpha} \longrightarrow N \boldsymbol{\lambda \alpha} = \boldsymbol{K \alpha}. \tag{3.7}$$

The eigenvalue problem needs to be solved for nonzero eigenvalues. The projected data set given by $\boldsymbol{\Phi}(x_i)$ needs to have a zero mean to centralize the data around the origin in $\boldsymbol{\mathcal{H}}$. The data points after centralizing is given by,

$$\widetilde{\boldsymbol{\Phi}}(x_i) = \boldsymbol{\Phi}(x_i) - \frac{1}{N} \sum_{k=1}^{N} \boldsymbol{\Phi}(x_k). \tag{3.8}$$

The corresponding elements of the Gram matrix are given by,

$$\widetilde{\boldsymbol{K}}_{ij} = \widetilde{\boldsymbol{\Phi}}(x_i)^T \widetilde{\boldsymbol{\Phi}}\big(x_j\big) \tag{3.9}$$

$$\begin{aligned} &= \boldsymbol{\Phi}(x_i)^T \boldsymbol{\Phi}\big(x_j\big) - \frac{1}{N} \sum_{k=1}^{N} \boldsymbol{\Phi}(x_i)^T \boldsymbol{\Phi}(x_k) - \frac{1}{N} \sum_{k=1}^{N} \boldsymbol{\Phi}(x_k)^T \boldsymbol{\Phi}\big(x_j\big) \\ &\quad + \frac{1}{N^2} \sum_{l=1}^{N} \sum_{k=1}^{N} \boldsymbol{\Phi}(x_l)^T \boldsymbol{\Phi}(x_k) \end{aligned} \tag{3.10}$$

$$= \boldsymbol{k}(x_i, x_j) - \frac{1}{N}\sum_{k=1}^{N}\boldsymbol{k}(x_i, x_k) - \frac{1}{N}\sum_{k=1}^{N}\boldsymbol{k}(x_k, x_j) + \frac{1}{N^2}\sum_{l=1}^{N}\sum_{k=1}^{N}\boldsymbol{k}(x_l, x_k). \tag{3.11}$$

The Matrix notation can be express as,

$$\widetilde{\boldsymbol{K}} = \mathbf{K} - \mathbf{1}_{\mathrm{N}}\mathbf{K} - \mathbf{K}\mathbf{1}_{\mathrm{N}} + \mathbf{1}_{\mathrm{N}}\mathbf{K}\mathbf{1}_{\mathrm{N}}, \tag{3.12}$$

where $\mathbf{1}_{\mathrm{N}}$ denotes $N \times N$ matrix in which every element takes on the value of $1/N$. The normalization condition for the coefficients of $\boldsymbol{\alpha}_m$ for $m = 1, 2, \ldots, M$ is obtained by requiring that the eigenvectors in feature space $\boldsymbol{\mathcal{H}}$ to be normalized,

$$1 = \|\boldsymbol{V}_m\|^2 = \boldsymbol{V}_m^T\boldsymbol{V}_m = \sum_{i=1}^{N}\sum_{j=1}^{N}\boldsymbol{\alpha}_{mi}\boldsymbol{\alpha}_{mj}\boldsymbol{\Phi}(x_i)^T\boldsymbol{\Phi}(x_j) \tag{3.13}$$

$$= \boldsymbol{\alpha}_m^T\boldsymbol{K}\boldsymbol{\alpha}_m = \boldsymbol{\lambda}_m N\boldsymbol{\alpha}_m^T\boldsymbol{\alpha}_m = \boldsymbol{\lambda}_m N\|\boldsymbol{\alpha}_m\|^2, \tag{3.14}$$

$$\|\boldsymbol{\alpha}_m\| = \frac{1}{\sqrt{\boldsymbol{\lambda}_m N}}. \tag{3.15}$$

To extract principal components corresponding to the kernel of test point x, we need to compute the projection of the test point onto the eigenvectors $\boldsymbol{V}_m$ in $\boldsymbol{\mathcal{H}}$ given by,

$$\boldsymbol{y}_m(x) = \boldsymbol{\Phi}(x)^T\boldsymbol{V}_m = \sum_{n=1}^{N}\boldsymbol{\alpha}_{mn}\boldsymbol{\Phi}(x)^T\boldsymbol{\Phi}(x_n) = \sum_{n=1}^{N}\boldsymbol{\alpha}_{mn}\boldsymbol{k}(x, x_n). \tag{3.16}$$

3.2 KPCA Anomaly Detection Method

According to Heiko Hoffmann, author of *Kernel PCA for Novelty Detection*, to detect anomalies, the reconstruction error in feature space can be used,

$$p(\widetilde{\Phi}) = (\widetilde{\Phi}\cdot\widetilde{\Phi}) - (\boldsymbol{W}\widetilde{\Phi}\cdot\boldsymbol{W}\widetilde{\Phi}) \tag{3.17}$$

where the matrix $\boldsymbol{W}$ contains L row vectors of $V_l = \sum_{n=1}^{N}\alpha_{ln}\Phi(x_n)$, L is number of eigenvectors and n is the number of samples. We need to eliminate $\widetilde{\Phi}$ to write the potential

as function of vector z in the original space. The projection $f_l(z)$ of $\widetilde{\Phi}(z)$ onto the eigenvector V_l needs to be evaluated using the kernel function.

$$f_l(z) = \widetilde{\Phi}(z) \cdot V_l$$

$$= \left(\Phi(z) - \frac{1}{N} \sum_{n=1}^{N} \Phi(x_n) \right) \cdot \left(\sum_{n=1}^{N} \alpha_{ln} \Phi(x_n) - \frac{1}{N} \sum_{n,j=1}^{N} \alpha_{ln} \Phi(x_j) \right) \tag{3.18}$$

$$= \sum_{n=1}^{N} \alpha_{ln} \left[k(z, x_n) - \frac{1}{N} \sum_{i=1}^{N} k(x_n, x_i) - \frac{1}{N} \sum_{j=1}^{N} k(z, x_j) + \frac{1}{N^2} \sum_{i,j=1}^{N} k(x_i, x_j) \right]$$

The potential in feature space can be represented as,

$$p(\widetilde{\Phi}) = (\widetilde{\Phi} \cdot \widetilde{\Phi}) - \sum_{l=1}^{L} f_l(z)^2 , \tag{3.19}$$

where the scalar product $(\widetilde{\Phi} \cdot \widetilde{\Phi})$ is the spherical potential which can be replace with $p_s(z)$,

$$p(z) = p_s(z) - \sum_{l=1}^{q} f_l(z)^2. \tag{3.20}$$

The spherical potential field $p_s(z)$ in feature space $\mathcal{H}$ is nothing more than the reconstruction error with no principal components. The potential of a point z in the original space is the square distance from the mapping $\Phi(z)$ to the center Φ_0. This can be written with the kernel function which gives the following expression for $p_s(z)$,

$$p_s(z) = \left\| \widetilde{\Phi}(z) \right\|^2 = \langle \Phi(z) - \Phi_0, \Phi(z) - \Phi_0 \rangle \tag{3.21}$$

$$= \langle \Phi(z), \Phi(z) \rangle - 2\langle \Phi(z), \Phi_0 \rangle + \langle \Phi_0, \Phi_0 \rangle \tag{3.22}$$

$$= \langle \Phi(z), \Phi(z) \rangle - \frac{2}{N} \sum_{n=1}^{N} \langle \Phi(z), \Phi_n \rangle + \frac{1}{N^2} \sum_{i,j=1}^{N} \langle \Phi_i, \Phi_j \rangle \tag{3.23}$$

$$= k(z,z) - \frac{2}{N}\sum_{n=1}^{N} k(z, x_n) + \frac{1}{N^2}\sum_{i,j=1}^{N} k(x_i, x_j), \tag{3.24}$$

where $\widetilde{\Phi}(z) = \Phi(z) - \Phi_0$ and $\Phi_0 = \frac{1}{N}\sum_{n=1}^{N}\Phi(x_n)$, which is used to center the data in the feature space. The reconstruction error in the original space can finally be represented as the follow,

$$p(z) = k(z,z) - \frac{2}{N}\sum_{n=1}^{N} k(z, x_n) + \frac{1}{N^2}\sum_{i,j=1}^{N} k(x_i, x_j) - \sum_{l=1}^{q} f_l(z)^2. \tag{3.25}$$

We can check if each new data point $\mathcal{Z}$ is anomalous by computing the reconstruction error and finding the maximum value over all nominal data points. The maximum reconstruction error value ε is known as the threshold value to detect anomalies. The following expression can show if the new data point value $\mathcal{Z}$ is anomalous,

$$p(\mathcal{Z}) \geq \varepsilon. \tag{3.26}$$

From this we can see which $\mathcal{Z}$ values are above the threshold ε to detect anomalies.

3.3 KPCA Simulation Results

The decision boundaries for Kernel PCA is demonstrated by using the data provided from Northrop Grumman. Northrop Grumman provided both nominal and off-nominal data that was used for this research. The KPCA algorithm was applied on each separate column of data. The main column that was focus on was column 6 because of how easy you can see the anomalies in the off-nominal data. The data had to be down sampled because of KPCA being to computational expensive to use on large data sets. To detect anomalies the threshold was calculated by finding the maximum reconstruction error from the nominal data. After finding the anomaly detection threshold it can then be applied on the off nominal data to detect anomalies as shown in figure 3.2.

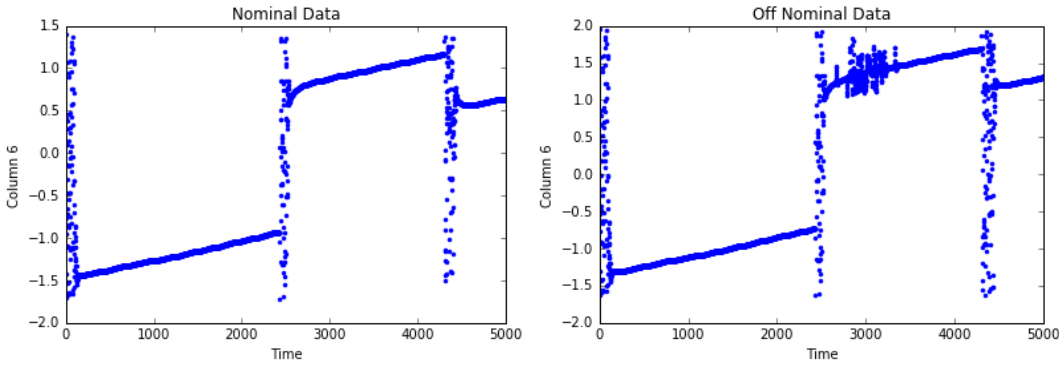

Figure 3.1 Nominal and Off Nominal Data for Column 6

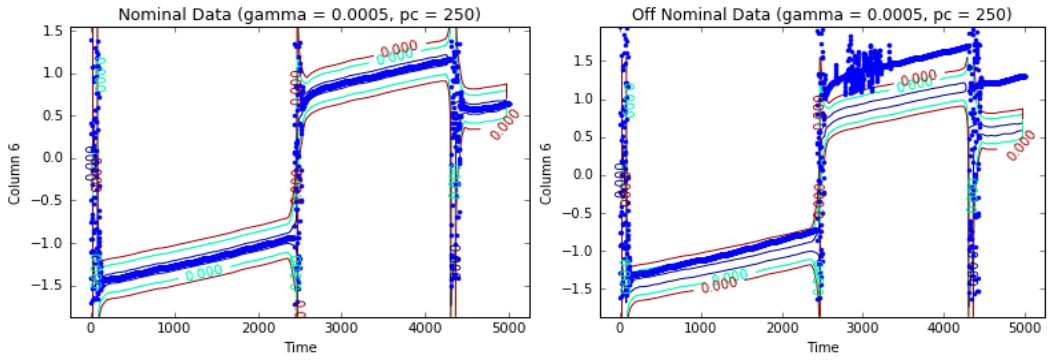

Figure 3.2 Decision Boundaries for KPCA on Nominal and Off Nominal Data

The problem of KPCA being too computational expensive arises from KPCA using a $N \times N$ Gram matrix, where N is the number of samples. Depending on the number samples you have in your data set the Gram matrix becomes more computationally expensive as the number of samples increases making KPCA very expensive to run. In standard PCA this problem does not exist becomes it uses a $M \times M$ covariance matrix, which just depends on the number of M features you have in your data set. The number of samples KPCA was limited to was 3000 samples, which is shown in figure 3.1.

One of the benefits of using kernel PCA is having the ability to choose the kernel function $\mathrm{K}(x, y)$. The kernel function is a similarity function that allows you to operate in a high dimensional feature space by simply computing the inner products of all the pairs

of data in the feature space. Some of the most common and widely used kernels include the Linear, Polynomial and Gaussian kernels, given by,

$$\text{Linear: } \mathrm{K}(x, y) = x \cdot y, \tag{3.27}$$

$$\text{Polynomial: } \mathrm{K}(x, y) = (1 + x \cdot y)^p, \tag{3.28}$$

$$\text{Gaussian: } \mathrm{K}(x, y) = e^{\frac{-|x-y|^2}{2\sigma^2}}. \tag{3.29}$$

The kernel function that was used for KPCA was the Gaussian kernel, because it can map the inputs onto the surface of an infinite-dimensional sphere.

By choosing to use a Gaussian kernel we introduce a new parameter σ^2, which is the variance of the Gaussian. The other parameter that KPCA uses is number of principal components. Increase and decreases both parameters can have a great effect on the threshold to detect anomalies. Having to change both parameters depending on the type of data you are running is another disadvantage with using KPCA.

A major problem that was discovered when implementing KPCA is that it can't handle time series data. KPCA can be trained with nominal data and a very accurate decision boundary can be created for that data as shown in figure 3.2, but when applying that same decision boundary to the off nominal data you don't know exactly when in time that same decision boundary is required. This algorithm can only handle non-sequential data that don't change over time. New methods that can handle time series data will be explored in the next sections, which includes a windowing and clustering approach, and Recurrent Neural Networks.

CHAPTER 4: K-MEANS CLUSTERING

The challenges of anomaly detection in time series can present many difficult problems. According to Ted Dunning and Ellen Friedman, authors of *Practical Machine Learning*, the problems presented with anomaly detection in time series can be solve with the methods of windowing and clustering. This technique involves extracting short sequences of the original signal in a way that these short sequences can be added back together to reconstruct the original signal. These short sequences that were extracted are used to form a dictionary of component shapes that can be to recognize the original signal.

4.1 Windowing Method

The first step of windowing is to split the waveform into overlapping segments, with the section of the original data sampled sliding along by two samples each time. This process was applied to the nominal data for column 6 displayed in figure 4.1. After windowing the entire original waveform, a dictionary of component shapes is form with each component shape having a sample length of 32 samples as shown in figure 4.2. The entire dictionary produced 9985 waveform segments. The waveform dictionary segments

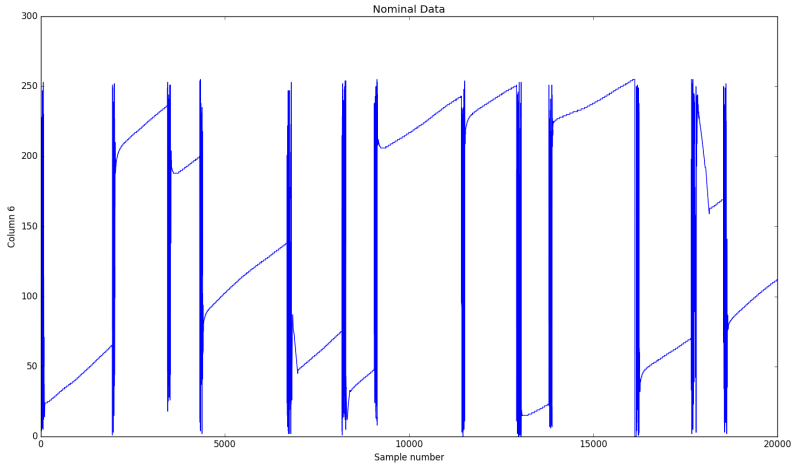

Figure 4.1 Nominal Data for Column 6

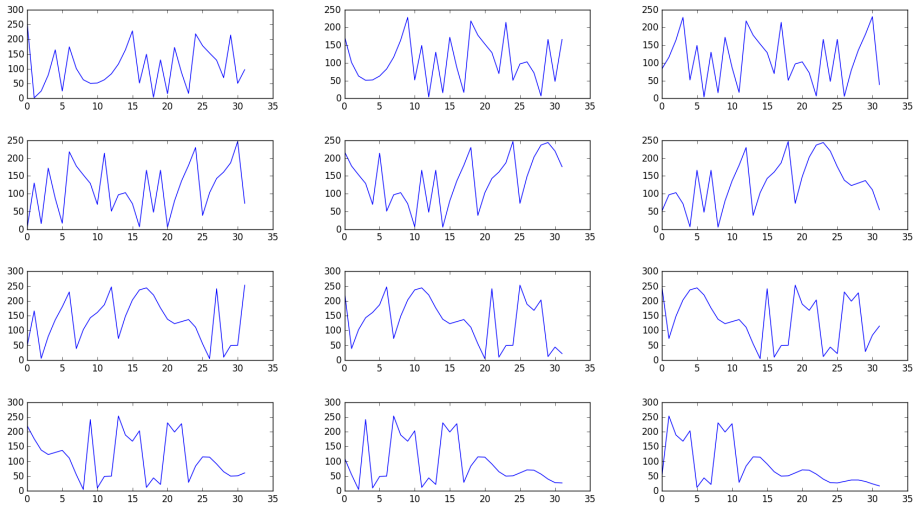

Figure 4.2 Dictionary of Component Shapes for Column 6

do not necessarily have to begin and end with a value of zero. The learned normal waveform segment will also have non-zero starts and end, which can cause problems when we try to reconstruct our original waveform by adding together our learned segment.

When adding each window segment together that don't begin and end with zero we'll end up with discontinuities in the reconstructed waveform. The way to avoid this problem is to apply a window function to the data, which forces the start and end to be zero for each waveform segment. The window function used on each waveform segment in the dictionary is shown in figure 4.3.

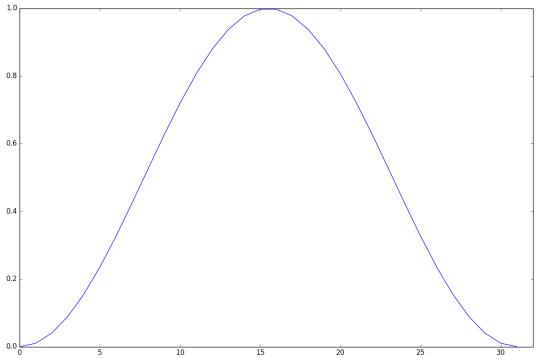

Figure 4.3 Window Function

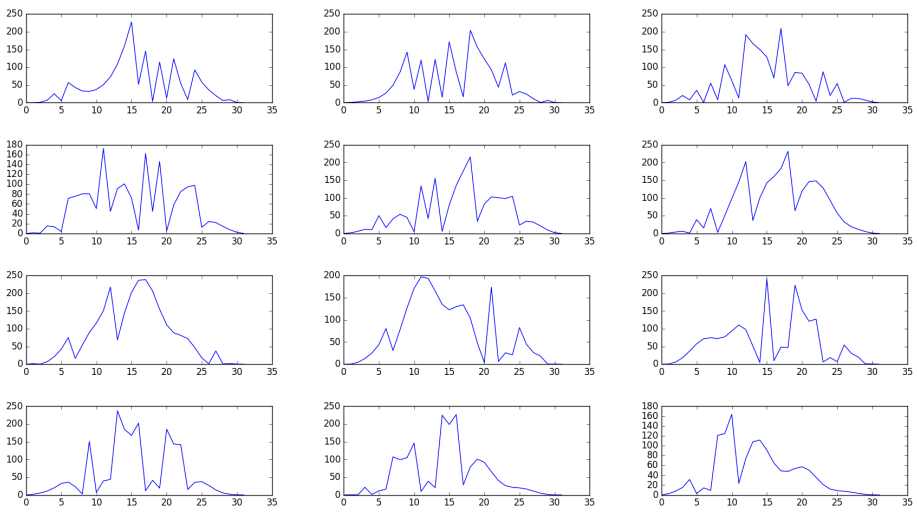

Figure 4.4 Window Function Filtered Dictionary of Component Shapes for Column 6

The window function can be applied to the data by multiplying each waveform segment found in the dictionary by the window function. We can see the difference this windowing process makes in figure 4.4. The segments are now flat at the start and end, which is perfect to be joined together later. Windowing also has the effect of making the segments less affected by the waveform either side of the segment. The waveform shape represented by the segment is now more concentrated in the middle.

4.2 Clustering

Building a waveform dictionary with windowing can lead to many component shapes being very similar with each other. These similarities between component shapes can be exploited by using clustering to build a catalog of a small number of basic shapes, with various horizontal translations. Clustering can find the most commonly used signal shapes for reconstructing a representation of the original signal. The waveform dictionary originally produced 9985 waveform segments from column 6, which can be significantly reduced by using a common clustering algorithm such as K-Means. Reducing the size of the dictionary is important to increase the performance of the anomaly detection algorithm.

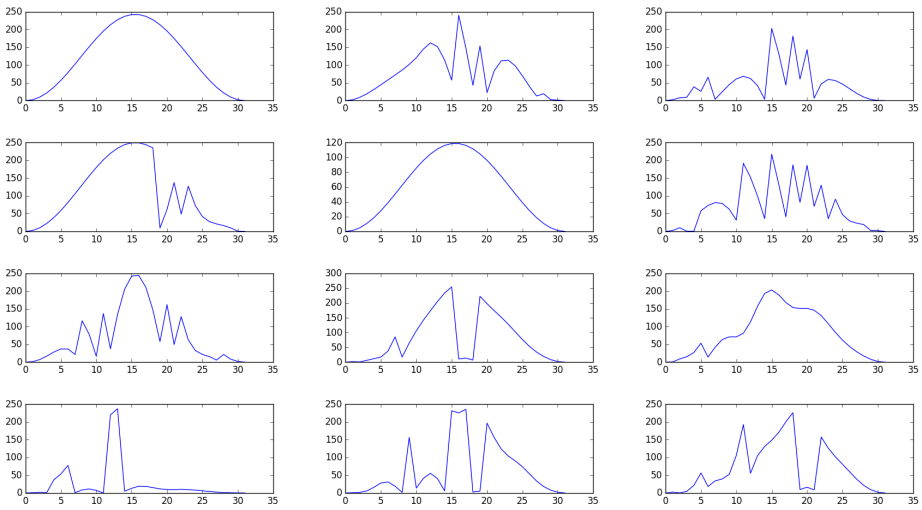

Figure 4.5 Clustered Dictionary Component Shapes for Column 6

4.3 K-Means Clustering Algorithm

To perform K-means clustering the number of k clusters first needs to be defined. The number of clusters that are selected will determine exactly how large your new waveform dictionary will become. The next step is to partition the training data points $x_1, x_2, \ldots, x_m$ in k clusters, where m is the number of samples. The goal is to assign a cluster to each data point. K-means is a clustering method that aims to find the positions $\mu_1, \mu_2, \ldots, \mu_K$ of the cluster centroids that minimize the distance from the data points to the cluster. K-means clustering solves the objective of minimizing the average squared Euclidean distance, which can be express as,

$$\arg\min_{c} \sum_{i=1}^{k} \sum_{j=1}^{c_i} \|x_j - \mu_i\|^2 \tag{4.1}$$

where c_i is the set of points that belong to the i^{th} cluster. By applying this clustering algorithm to the waveform dictionary the has been significantly reduced to keep only the most important basic shapes that is required to reconstruct the original signal. This newly formed dictionary is presented in figure 4.5.

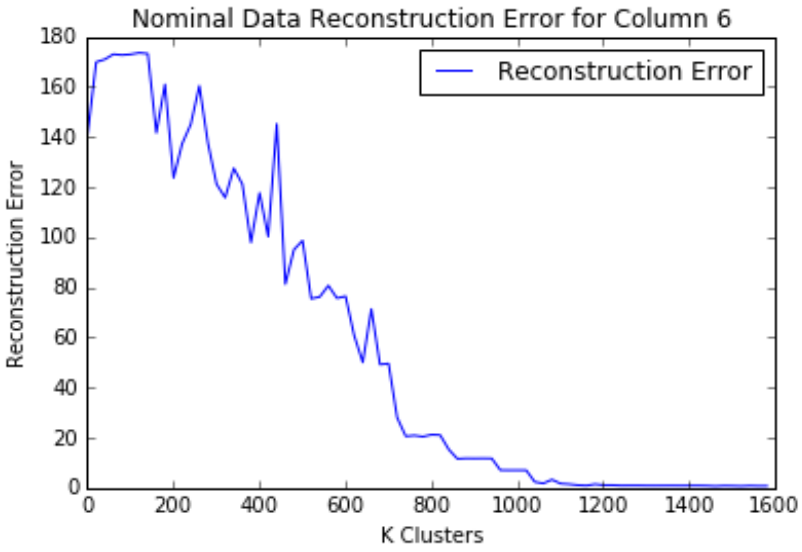

Figure 4.6 Nominal Data Reconstruction Error for Column 6

4.4 Reconstruction from Clusters

To build an ideal model to reconstruct the original waveform without any errors. The reconstruction of the original signal needs to be tested to verify that our waveform dictionary can recreate the original signal.

The process first starts with splitting the data into overlapping segments, which will allow us to reconstruct the original signal. Second, we need to find the cluster centroid which best matches our segment and then use that centroid as the reconstruction for that segment. After doing this procedure for each segment all the reconstruction segments can be joined together to form the reconstruction of the original signal.

If a reconstruction error was found during the verification test to reconstruction the original signal, then our waveform dictionary doesn't have enough components to correctly reconstruct the signal. To solve this problem, the number of k clusters needs to be increase until the maximum reconstruction error goes to zero. The maximum reconstruction error reaches zero for column 6 at around 1100-1200 clusters as shown in figure 4.6. If the

waveform dictionary doesn't have these many components after clustering, then it can't fully reconstruct the original signal correctly.

4.5 K-Means Clustering Anomaly Detection Simulation Results

The reconstruction of various signals was tested with the waveform dictionary created from the nominal data provided by Northrop Grumman. The waveform dictionary must be able to reconstruction of the nominal data as close as possible before using the dictionary with off nominal data. If the reconstruction error from the tested signals are very large, then those signals must be anomalous, since it doesn't match the original signals.

The first signal that was tested is column 6 as shown in figure 4.7. In figure 4.8 we can verify that the original signal could reconstruct itself by using 1150 clusters for the waveform dictionary. When testing the off nominal data, the waveform dictionary had a difficult time reconstructing the signal where there were short impulse spikes. This would give us large reconstruction error wherever there was any type of small impulse spike that should not get reconstructed. To solve this problem, an anomaly detector was created using a sliding window that would check if the reconstruction error was above a set threshold for a set number of samples in the sliding window.

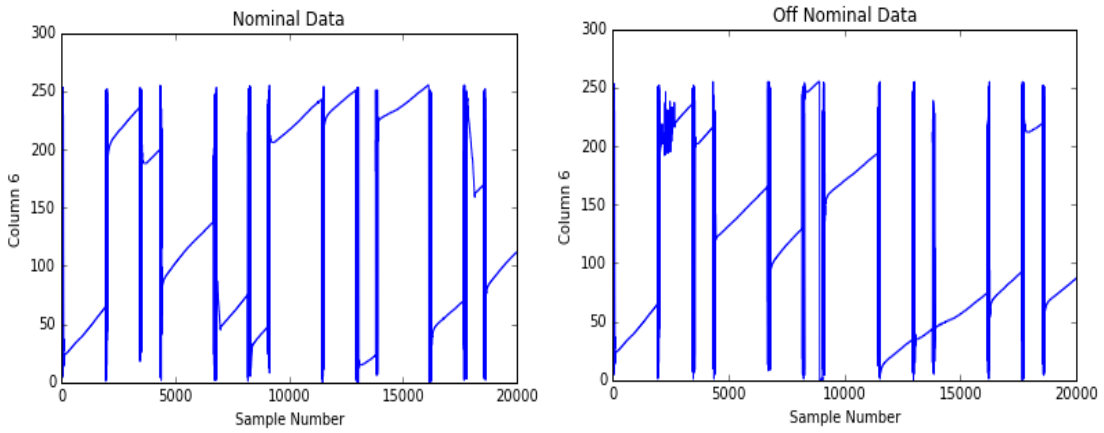

Figure 4.7 Nominal and Off Nominal Data for Column 6

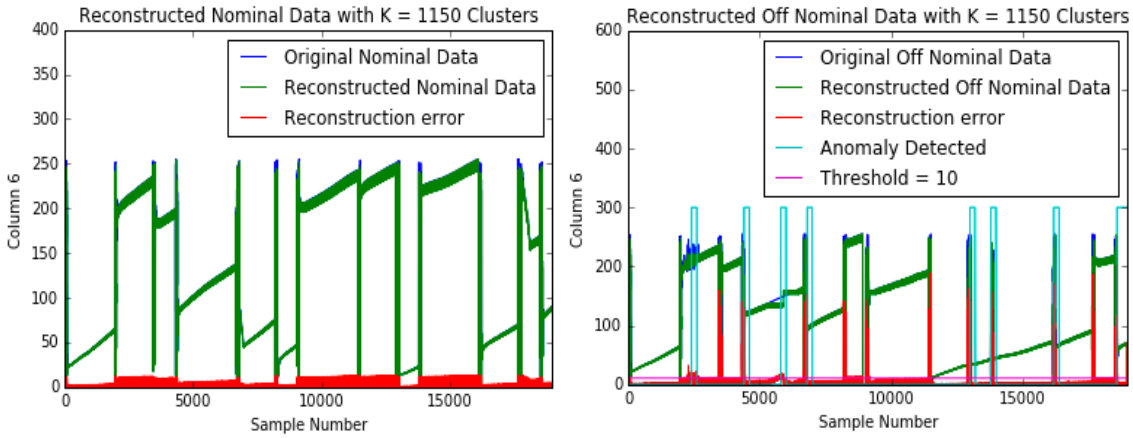

Figure 4.8 Reconstructed Nominal and Off Nominal Data for Column 6

The challenges that occurred when implementing the anomaly detector are that various parameters needs to be set correctly to reduce the number of false alarms. The parameters that were used are the window size, sliding window step, threshold, and threshold flag size. In column 6 the parameters that were set for the anomaly detector was a window size of 400 samples, sliding window step of 10 samples, threshold of the 90th percentile of the reconstruction error, and threshold flag size of 260 window samples. The sliding window anomaly detector for column 6 is shown in figure 4.8.

The second signal that was tested was column 5 as shown in figure 4.9. The number of clusters needed to perfectly reconstruct the original signal is significantly larger then column 6. To reconstruct the original signal 2800 clusters were needed for the waveform dictionary as shown in figure 4.10. The same parameters for the sliding window anomaly detector used for column 6 was also used for column 5. The anomalies detected and reconstruction of the off nominal data can be observed in figure 4.10. The accuracy of the sliding window anomaly detector can't be determined exactly, since the location of the anomalies are unknown in the off nominal data. New testing data with the anomalies locations would be very beneficial to test the accuracy and precision of our algorithms.

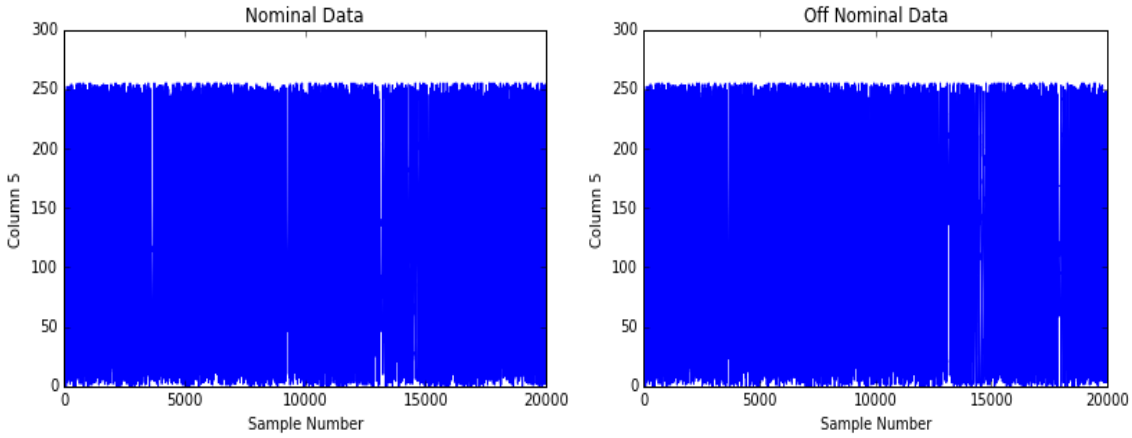

Figure 4.9 Nominal and Off Nominal Data for Column 5

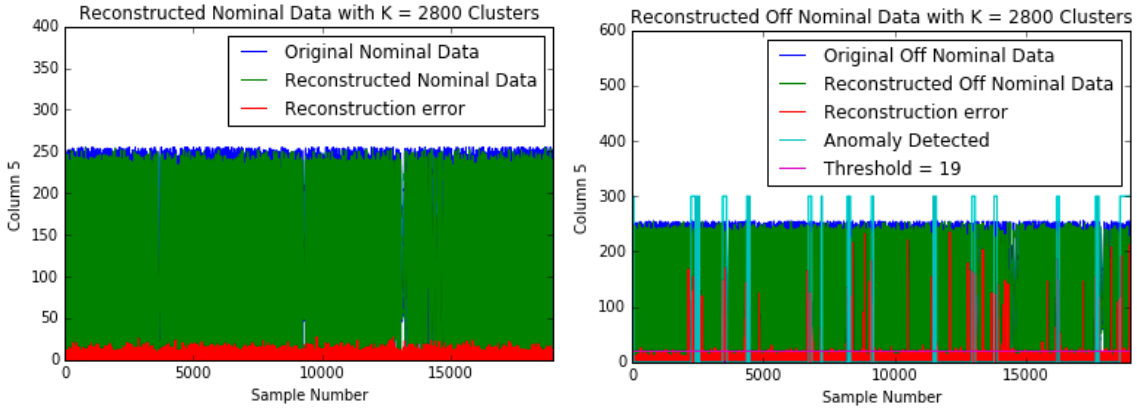

Figure 4.10 Reconstructed Nominal and Off Nominal Data for Column 5

CHAPTER 5: RECURRENT NEURAL NETWORKS

Recurrent Neural Networks (RNNs) is a popular time series model that can handle sequential information. RNNs are very useful for sequential information because they can perform the same task for every element of a sequence, with the output being depended on the previous computations. In a standard neural network, all inputs and outputs are assumed to be independent of each other, which is not the case for RNNs. RNNs can be thought of as multiple copies of the same neural network in time that passes information from the previous neural network in time. This can form a short or long term memory for the neural network depending on how far back it is looking.

A simple standard recurrent neural network can be represented with the following equations,

$$h_t = \tanh(\theta_{xh} x_t + \theta_{hh} h_{t-1} + b_h) \tag{5.1}$$

$$o_t = \theta_{ho} h_t + b_o, \tag{5.2}$$

where x_t represents the current input, h_t the current hidden states, o_t the current output, and θ stands for the weights parameters for the Recurrent Neural Network. A diagram that represents a recurrent neural network is shown below in figure 5.1.

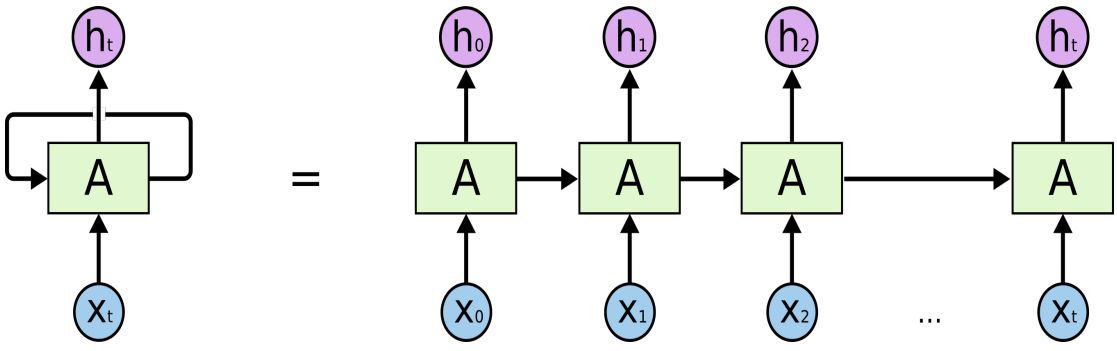

Figure 5.1 Recurrent Neural Network [13]

To define the loss or error of the RNN the cross-entropy loss needs to be calculated by the following equation,

$$E(o, y) = -\sum_{t} y_t \log(o_t), \tag{5.3}$$

where o_t is the current output and y_t is the target output [14]. When calculating the loss, we treat the full RNN sequence as one training example. This allows the total error to just be the sum of the errors at each time step.

5.1 Long-Term Dependencies Problem

RNNs present a major problem when dealing with long-term dependencies. The idea behind RNNs is to be able to connect previous information from the past to solve a present problem. By looking at too much relevant information from the past, RNNs start to be unable to learn how to connect the previous information. This problem can make RNNs suffer from a vanishing and exploding gradient that makes learning long range dependencies very difficult.

To train RNNs we need to find the weight parameters of θ that minimizes the total loss on the training data. This is achieved by performing backpropagation through time (BPTT), which is a gradient-based technique. BPTT learning algorithm is an extended version of backpropagation algorithm used for feed forward neural networks. The BPTT algorithm involves unfolding the RNN so that the recurrent weights are shared and duplicated for each number of time steps.

The most common gradient-based technique used is a Stochastic Gradient Descent (SGD). SGD is performed by iterating over all training data and during each iteration the weight parameters are push into the direction that reduces the loss. The direction of the loss

is given by the gradients of the loss, which are the sum of all gradients at each time step for one training example. The sum of the gradients can be shown as the following equation,

$$\frac{\partial E}{\partial \theta} = \sum_{t} \frac{\partial E_t}{\partial \theta} \ . \tag{5.3}$$

To calculate these gradients, the chain rule of differentiation needs to be used, which turns the equation into the following,

$$\begin{aligned} \frac{\partial E_t}{\partial \theta} &= \sum_{k=1}^{t} \frac{\partial E_t}{\partial o_t} \frac{\partial o_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial \theta} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial o_t} \frac{\partial o_t}{\partial h_t} \left(\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial \theta} \\ &= \sum_{k=1}^{t} \frac{\partial E_t}{\partial o_t} \frac{\partial o_t}{\partial h_t} \left(\prod_{i=k+1}^{t} \theta^T diag[\phi(h_{i-1})] \right) \frac{\partial h_k}{\partial \theta}, \end{aligned} \tag{5.4}$$

where ϕ is the activation function tanh or sigmoid and $\frac{\partial h_t}{\partial h_k}$ takes the form of a product of $t-k$ Jacobian matrices [10]. We can see that the norm of the Jacobian is bounded to the following condition,

$$\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| \leq \|\theta^T\| \|diag[\phi(h_{i-1})]\| \leq \gamma_\theta \gamma_\phi, \tag{5.5}$$

$$\frac{\partial h_t}{\partial h_k} \leq \left(\gamma_\theta \gamma_\phi\right)^{t-k}, \tag{5.6}$$

where if the constant $\gamma_\theta \gamma_\phi$ is larger than one the gradient will explode or if the constant $\gamma_\theta \gamma_\phi$ is smaller than one the gradient will vanish [10]. By performing BPTT to train the RNN we can see that the two major problems that occur are the vanishing and exploding gradients.

5.2 Long Short Term Memory Networks

Long Short Term Memory (LSTM) networks is a special RNN that are capable of learning long term dependencies. LSTM is design specifically to solve the long-term

dependency problem that occurs in standard Recurrent Neural Networks. In a standard RNN, the repeating neural network cells are a single tanh layer as shown in figure 5.2. In this model, the RNN will suffer from the vanishing and exploding gradient problem. To solve this problem, the repeating neural network cells found in the standard RNN model can be replaced with LSTM cells as shown in figure 5.3. The LSTM cell uses a 'memory cell' to recall past information only when needed thereby working around the vanishing gradient problem.

The LSTM cell is a four-layer neural network, which consist of three types of gates the input gate (i_t), output gate (o_t), and forget gate (f_t). The input gate scales the input to the cell, which is the write function of the LSTM cell. The output gate scales the output from the cell, which is the read function of the LSTM cell. The forget gate scales old cell value, which is the reset function of the LSTM cell. These three gates can remove or add information to the cell as needed.

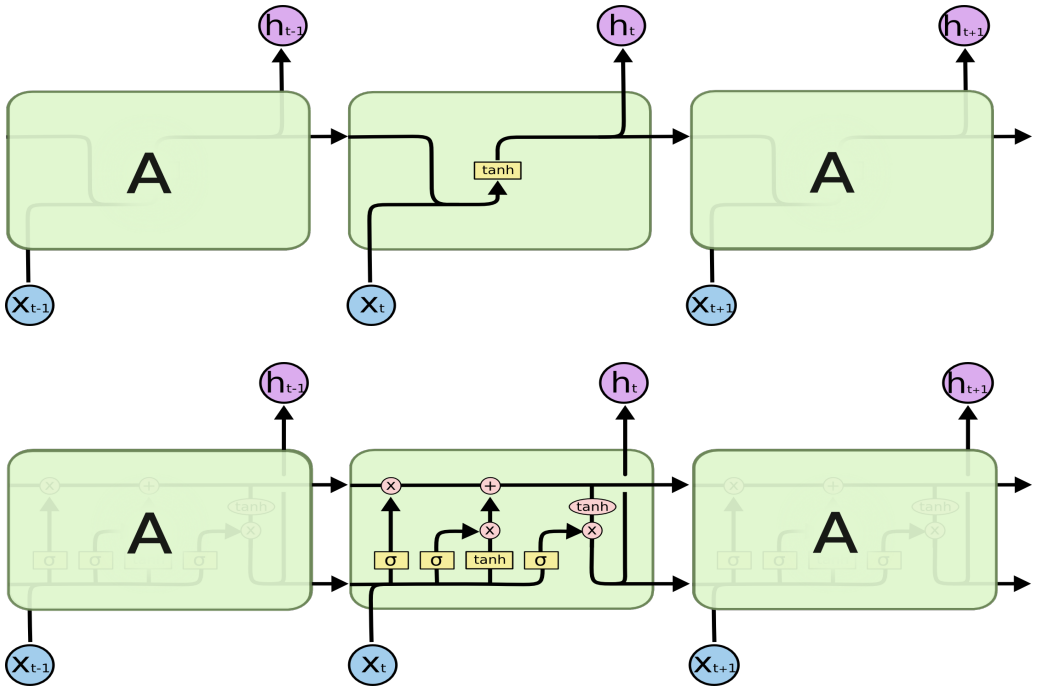

Figure 5.2 Standard RNN (Top) and LSTM RNN (Bottom) [13]

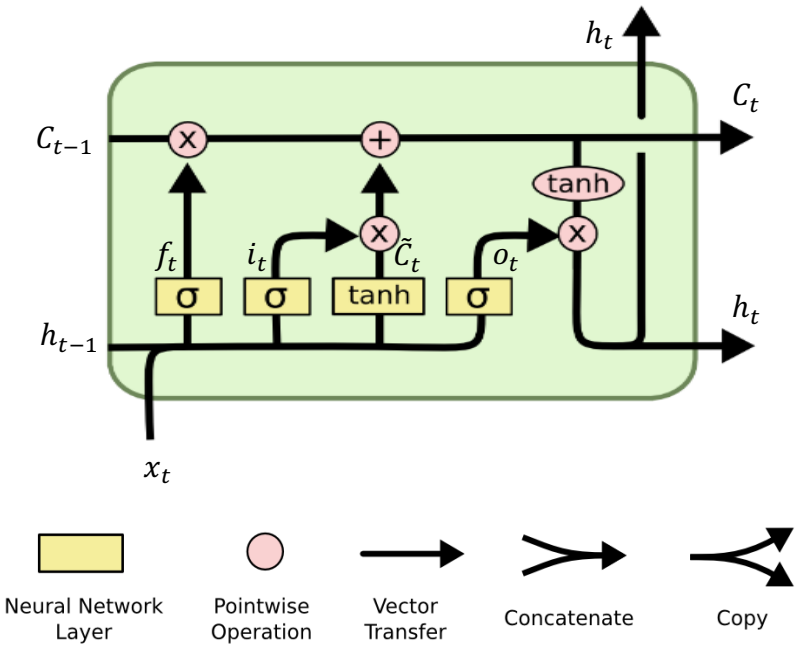

Figure 5.3 Long Short Term Memory (LSTM) Cell [13]

The LSTM cell shown in figure 5.4 can be implemented with the following equations below,

$$f_t = \sigma\left(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f\right), \tag{5.7}$$

$$i_t = \sigma(\theta_{xi} x_t + \theta_{hi} h_{t-1} + b_i), \tag{5.8}$$

$$\tilde{C}_t = tanh(\theta_{xC} x_t + \theta_{hC} h_{t-1} + b_C), \tag{5.9}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t, \tag{5.10}$$

$$o_t = \sigma(\theta_{xo} x_t + \theta_{ho} h_{t-1} + b_o), \tag{5.11}$$

$$h_t = o_t \odot tanh(C_t), \tag{5.12}$$

where σ stands for the logistic sigmoid function, $\odot$ is elementwise multiplication, and θ stands for the weight parameters [13]. These equations can be used to define each memory cell in the LSTM network.

5.3 RNN Anomaly Detection Method

The method to detect anomalies with RNN is to set up the RNN as a autoencoder RNN. This is a specific form of RNN that allows the trained RNN to reproduce its input at the output layer. Once the RNN learns how to reconstruct the original signal with no anomalies, it can then try to reconstruct a signal with anomalies. The reconstruction error that is found from trying to reconstruct the signal can be used to evaluate if an anomaly is present or not. The following procedure describes this process fully:

1. Sample data with sliding window method.
2. Setup the RNN model.
3. Train the RNN with RMSprop.
4. Optimize the number of layers and size of each layer.
5. Evaluate the anomaly scores with the reconstruction error.

5.4 RNN Anomaly Detection Simulation Results

One of the major obstacles that occurred while training RNNs were their training computational time. The method that was used to reduce RNNs training time was down sampling. The original data provided from Northrop Grumman was 200,000 samples. These samples were down sample by $1/10^{th}$ to reduce the number of samples of the original data to 20,000 samples. This is shown in figure 5.4 for the nominal and off nominal data of column 6.

Even with down sampling the original data, RNNs were still to computational expensive to trained because of other parameters such as sequential memory, number of layers stacked, and number of hidden states per layer. Unlike a typical neural network RNNs have sequential memory. Depending on how large the RNNs sequential memory

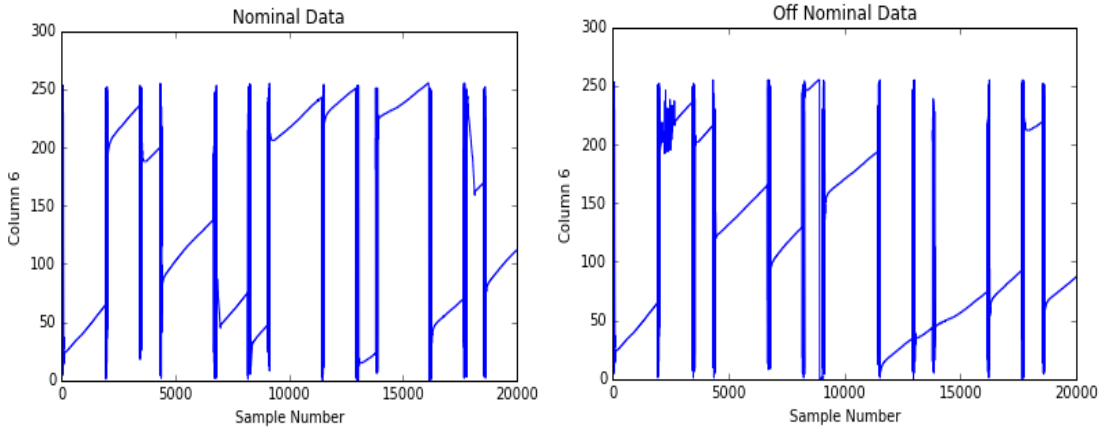

Figure 5.4 Nominal and Off Nominal Data for Column 6

can affect the computational time it takes to train. This is another factor that needs to be considered while setting up the parameters for the RNN. RNN layers can be stacked on top of each other to perform deep learning. By stacking the RNN layers this can dramatically increase the training time needed. Each of the RNN layers can have a different number of hidden states assigned. If the number of the hidden states per layer increase, then the computational time for training the RNN increases as well. These different parameters for RNN can greatly influence the training time.

To solve this problem, only the first 3000 samples of column 6 were used to train the RNN. The first 3000 samples were chosen because there was a visible anomaly in that section of the data. Since the computational time for training RNNs can become quite large by having a huge amount of sequential memory only 50 samples were used for the RNN memory. Only a small number of hidden states assigned to each layer could be used since the training time increases considerably if the hidden states are large.

First a two-layer stacked RNN was tested with the Northrop Grumman data. The number of hidden states that were chosen for each layer were keep low to keep the training time required as low as possible. One of the downsides of using less hidden states for our

Table 5.1 Two-Layer Stacked RNN

First Layer	128 hidden states
Second Layer	256 hidden states
Epochs	300
Memory	50 samples
Loss	0.0016
Run Time	12.85 Hours

RNN model is that it learns less features. By having the RNN learn less features the original signal might not be able to recreate the original signal completely with the decrease number of features available to reconstruct the signal. All the parameters for the two-layer stacked RNN are displayed in table 5.1.

The graph shown in figure 5.5 shows how well the RNN can reconstruct the original signal of a two-layer stacked RNN. The graph on the very top of the figure displays the first 3000 samples of the nominal data from column 6. The predicted signal of the original signal is in the middle section of figure 5.5. The RNN was trained for 300 Epochs, but still had a small amount of mean square error. This error is shown on the bottom of figure 5.5.

In figure 5.6 the graphs show how well the trained RNN performs on the tested signal with anomalies. The graph on the top shows the off-nominal data that contains anomalies in column 6. The predicted signal that is in the middle section of figure 5.6 couldn't predict the original signal perfectly since the RNN could not reconstruct the original signal perfectly with no error. We can see that the error from predicting the nominal data still appears in predicting the off nominal data. To be able to predict the anomalies in the off nominal data the nominal data needs to be predicted perfectly.

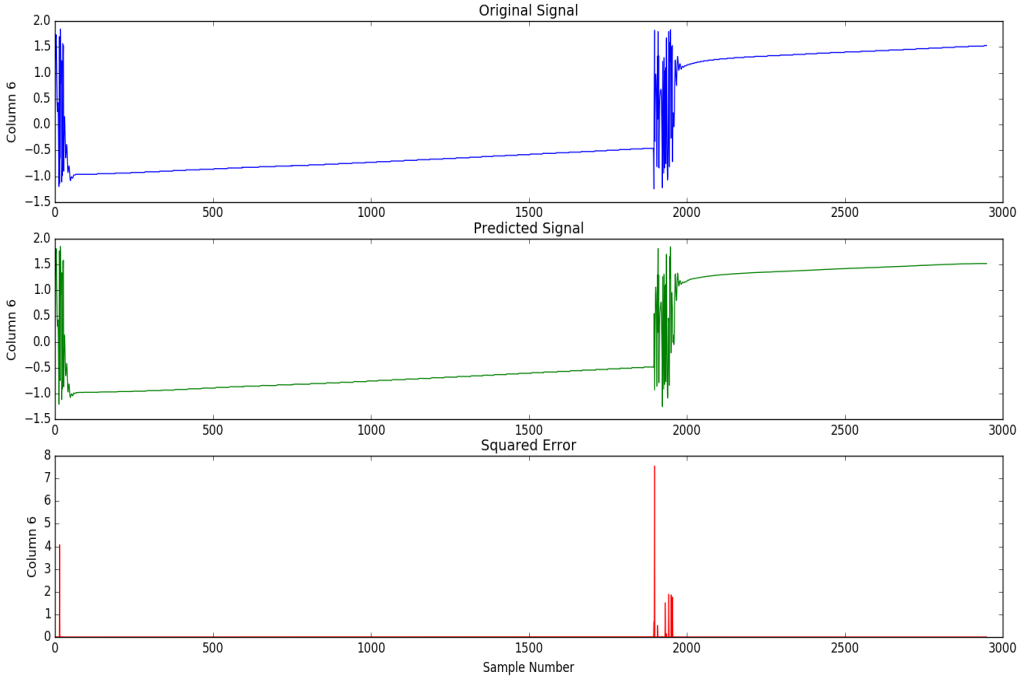

Figure 5.5 Original Signal (Top) Predicted Signal (Middle) Square Error (Bottom) for Two-Layer Stacked RNN

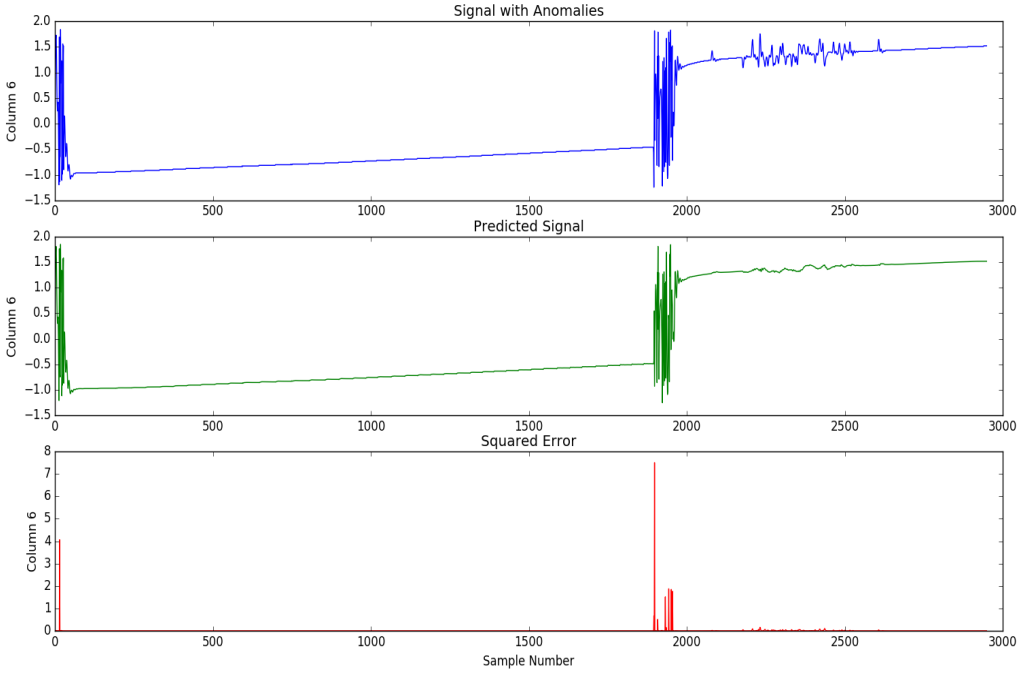

Figure 5.6 Signal with Anomalies (Top) Predicted Signal (Middle) Square Error (Bottom) for Two-Layer Stacked RNN

Table 5.2 Three-Layer Stacked RNN

First Layer	64 hidden states
Second Layer	512 hidden states
Third Layer	128 hidden states
Epochs	300
Memory	50 samples
Loss	0.0045
Run Time	27.5 Hours

Finally, a three-layer stacked RNN was tested with the Northrop Grumman data. The three-layer stacked RNN took longer to run then the two-layer stacked RNN since it had more layers to compute and hidden states. The same number of samples, memory, and epochs were chosen as the two-layer stacked RNN. All the parameters for the three-layer stacked RNN are displayed in table 5.2.

The graph shown in figure 5.7 shows how well the RNN can reconstruct the original signal of a three-layer stacked RNN. The three-layer stacked RNN did not train as well as the two-layer stacked RNN. The loss of the three-layer stacked RNN was 0.0045 while the loss of the two-layer RNN was 0.0016. The error of the three-layer stacked were worst then the two-layer stacked RNN. The three-layer stacked RNN was tested with the off nominal data in figure 5.8. The predicted signal did not improve from the previous two-layer stacked RNN since the three-layer stacked RNN predicted the nominal data worst. By increasing the number of layers for the RNN the predication can either get worst or better, but in our case the prediction got worst. We can clearly see from figure 5.8 that the two-layer stacked RNN perform better than the three-layer stacked RNN.

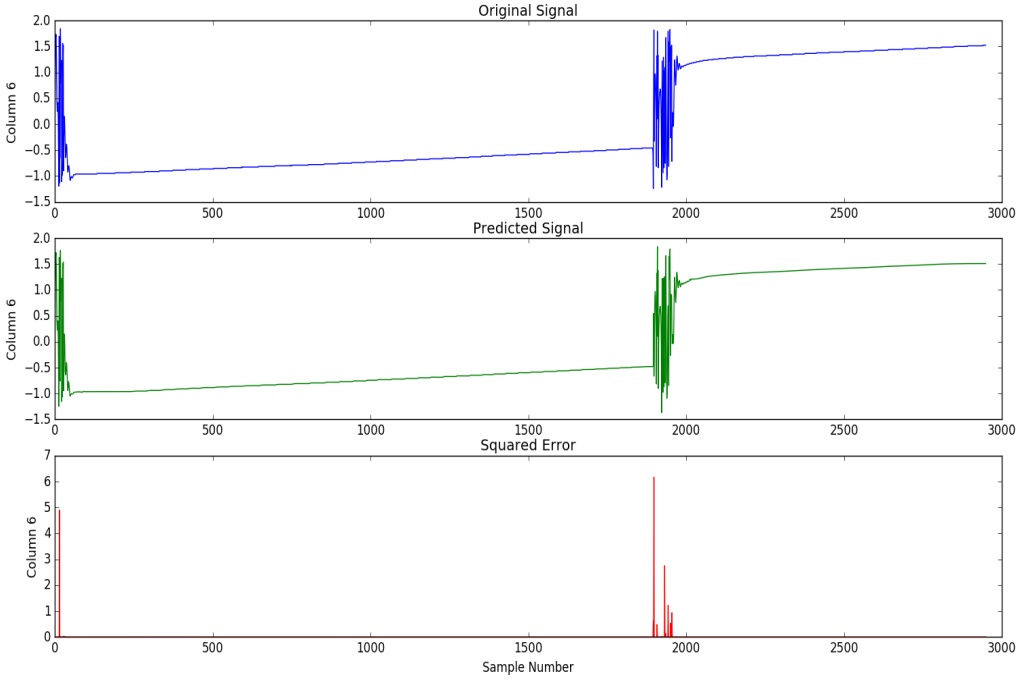

Figure 5.7 Original Signal (Top) Predicted Signal (Middle) Square Error (Bottom) for Three-Layer Stacked RNN

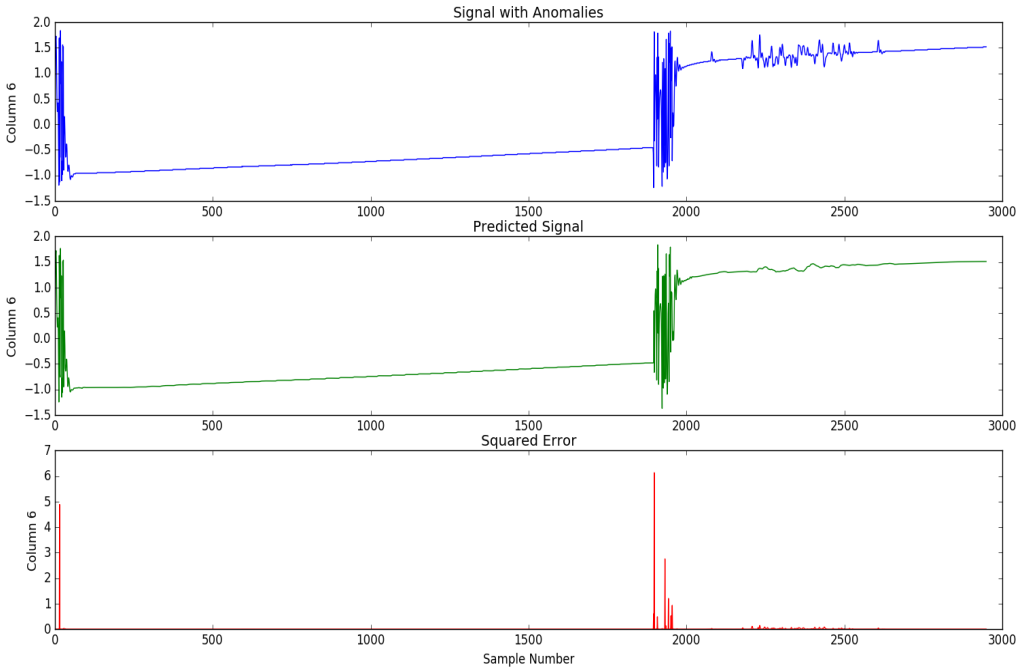

Figure 5.8 Signal with Anomalies (Top) Predicted Signal (Middle) Square Error (Bottom) for Three-Layer Stacked RNN

CHAPTER 6: CONCLUSION

Machine learning is a very powerful tool for anomaly detection. We could see that there were many different types of machine learning algorithms for anomaly detection that yield different results. The machine learning algorithms that were explored in this research were Kernel Principal Component Analysis (KPCA), K-Means Clustering, and Recurrent Neural Networks (RNN). Each of these algorithms offered diverse results to detect anomalies.

The results from each machine learning algorithm had various success and problems for detecting anomalies. KPCA did a great job at drawing decision boundaries around the nominal data to detect anomalies. The KPCA algorithm came into issues for handling time series data and computational power restriction. This algorithm could only handle static data that don't change over time.

K-means Clustering was used to solve the time series problem that KPCA couldn't solve. Window waveform segments was used to save the nominal data in a dictionary. This dictionary would allow the K-Means Clustering algorithm to handle sequential data unlike the KPCA algorithm. By windowing and clustering both the nominal and off nominal data anomalies can be detected with the reconstruction error. The reconstruction error was found by comparing the nominal and off nominal clustered dictionaries with each other to find the best match of each cluster centroid in the off nominal data. The results from this type of anomaly detection was a great improvement from the KPCA anomaly detection except for the false positives that occur on the impulse spikes of the signal.

Another alternative anomaly detection technique that can solve the time series problem is Recurrent Neural Networks. Recurrent Neural Networks were implemented by

training the RNN model with the nominal data to perfectly reconstruct the original signal. The most important part of using RNN for anomaly detection is to get a perfect reconstruction of the nominal data. If the reconstruction error of the original signal is not perfect, then false positives would occur while testing the off-nominal data for anomalies. One benefit of using RNN is that after training the nominal data the weight parameters can be saved, which can allow the testing of off nominal data to be done very quickly and efficiently.

The most difficult problem with Recurrent Neural Networks is that training the RNN model is very computationally expensive and time consuming. The amount of computationally power and time to train the RNN is affected by the number of samples, hidden features, memory, and stack layers for the RNN. The RNN model that was train for this research was never able to completely reconstruct the original signal due to the lack of computing power and time it takes to train the model. This problem leads to the RNN model having false positives when testing the off nominal data.

6.1 Future Work

Many improvements can be made to the machine learning anomaly detection systems researched. The accuracy and performance of these anomaly detection systems could be tested with evaluation metrics such as the f measure score and receiver operating characteristic curve. During this research, the location of the anomalies was not provided so we could not fine tune the algorithms to decrease the number of false positives detected. Another problem that was encountered during this research was the computational power required to run machine learning algorithm such as KPCA and RNNs. To get better results from machine learning algorithms that are computationally intensive, supercomputers with

powerful GPUs are needed to train the data. The parameters that were used for these algorithms were limited due to the computing capabilities of our computers. To be able to successfully train the nominal data the computing power used to train our systems needs to be addressed. These changes would greatly improve the results for KPCA and RNNs.

REFERENCES

[1] I.T. Jolliffe, "Principal Component Analysis," 2nd Edition, Springer, 2002.

[2] B. Schölkopf, A. J. Smola, "Learning with Kernels," MIT Press, Cambridge, MA, 2002.

[3] S. Mika, B. Schölkopf, A. J. Smola, K.-R. Müller, M. Scholz, G. Rätsch, "Kernel PCA and De-Noising in Feature Spaces," Advances in Neural Information Processing Systems, Vol. 11, pp. 536-542, Berlin, Germany, 1999.

[4] B. Schölkopf, A. J. Smola, K.-R. Müller, "Kernel Principal Component Analysis," Artificial Neural Networks – ICANN'97, Berlin, Germany, 1997.

[5] H. Hoffmann, "Kernel PCA for Novelty Detection," Max Planck Institute for Human Cognitive and Brain Sciences, Munich, Germany, 2007.

[6] T. Dunning, E. Friedman, "Practical Machine Learning - A New Look at Anomaly Detection," O'Reilly Media, Inc., 2014.

[7] C. M. Bishop, "Pattern Recognition and Machine Learning," Springer, 2006.

[8] I. Goodfellow, Y. Bengio, A. Courville, "Deep Learning," MIT Press, 2016.

[9] Y. Bengio, P. Simard, P. Frasconi, "Learning Long-Term Dependencies with Gradient Descent is Difficult," IEEE Transactions on Neural Networks, Vol. 5, No. 2, 1994.

[10] R. Pascanu, T. Mikolov, Y. Bengio, "On the difficulty of training recurrent neural networks," ICML (3), Vol. 28, pp. 1310-1318, 2013.

[11] S. Hochreiter, J. Schmidhuber, "Long Short-Term Memory," Neural Computation, Vol. 9, pp. 1735-1780, 1997.

[12] J. Schmidhuber, F. A. Gers, "Recurrent Nets that Time and Count," International Joint Conference on Neural Networks (IJCNN), Vol. 3, pp. 3189, 2000.

[13] C. Olah, "Understanding LSTM Networks," Colah's Blog, Aug. 27, 2015, http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[14] D. Britz, "Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients," Wildml AI, Deep Learning, NLP, Oct. 8, 2015, http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/

APPENDIX A: DATA COMPRESSION PRINCIPAL COMPONENT ANALYSIS MATLAB CODE

```
function [X_norm, mu, sigma] = featureNormalize(X)
        mu = mean(X);
        X_norm = bsxfun(@minus, X, mu);
        sigma = std(X_norm);
        X_norm = bsxfun(@rdivide, X_norm, sigma);
end
function [U, S] = pca(X)
        [m,n] = size(X);
        U = zeros(n);
        S = zeros(n);
        Sigma = (1/m)*(X')*X;
        [U, S, ~] = svd(Sigma);
end
function [VR,ASPE,TV] = varianceRetained(S,K)
        [m,~] = size(S);
        ASPE = 0;
        TV = 0;
        for i = 1:m
            if i <= K
                ASPE = S(i,i) + ASPE;
            end
            TV = S(i,i) + TV;
        end
        VR = ASPE/TV;
end
function Z = projectData(X, U, K)
        Z = zeros(size(X, 1), K);
        Ureduce = U(:,1:K);
        Z = X*Ureduce;
end
function X_rec = recoverData(Z, U, K)
        X_rec = zeros(size(Z, 1), size(U, 1));
        Ureduce = U(:,1:K);
        X_rec = Z*Ureduce';
end
function [X] = featureDenormalize(X_norm, mu, sigma)
        X_sigma = bsxfun(@times, X_norm, sigma);
        X = bsxfun(@plus, X_sigma, mu);
end

clc; clear; close all;
fprintf('Data Compression Example.\n\n');
varRetained = 0.99;
gaussianNoiseMean = 0.02;
gaussianNoiseVariance = 0.005;
% Load Chandra and Zeki Image
FileName1 = 'Advisor-Chandra.JPG';
FileName2 = 'Advisor-Zeki.JPG';
fprintf(1, 'Now reading %s\n', FileName1);
imageMatrix1 = imread(FileName1);
imageMatrix1 = rgb2gray(imageMatrix1);
imageMatrix1 = im2double(imageMatrix1);
fprintf(1, 'Now reading %s\n', FileName2);
imageMatrix2 = imread(FileName2);
imageMatrix2 = rgb2gray(imageMatrix2);
imageMatrix2 = im2double(imageMatrix2);
figure(1);
subplot(1, 2, 1);
imshow(imageMatrix1);
title('Original Chandra Image');
subplot(1, 2, 2);
imshow(imageMatrix2);
title('Original Zeki Image');
fprintf('The data of Chandra image has a size of: ');
fprintf('%d ', size(imageMatrix1));
fprintf('\nThe data of Zeki image has a size of: ');
fprintf('%d ', size(imageMatrix2));
fprintf('\n\nProgram paused. Press enter to continue.\n');
pause;
```

```
% Add noise to image
noiseImageMatrix1 =
imnoise(imageMatrix1,'gaussian',gaussianNoiseMean,gaussianNoiseVariance);
noiseImageMatrix2 =
imnoise(imageMatrix2,'gaussian',gaussianNoiseMean,gaussianNoiseVariance);
figure(2);
subplot(1, 2, 1);
imshow(noiseImageMatrix1);
title('Original Chandra Image with Noise');
subplot(1, 2, 2);
imshow(noiseImageMatrix2);
title('Original Zeki Image with Noise');
fprintf('\n\nProgram paused. Press enter to continue.\n');
pause;
[X_norm1, mu1, sigma1] = featureNormalize(noiseImageMatrix1);
[X_norm2, mu2, sigma2] = featureNormalize(noiseImageMatrix2);
figure(3);
subplot(1, 2, 1);
imshow(X_norm1);
title('Normalized Chandra Image with noise');
subplot(1, 2, 2);
imshow(X_norm2);
title('Normalized Zeki Image with noise');
fprintf('\n\nProgram paused. Press enter to continue.\n');
pause;
% Run PCA
[U1, S1] = pca(X_norm1);
[U2, S2] = pca(X_norm2);
fprintf('The data U1 has a size of: ');
fprintf('%d ', size(U1));
fprintf('\nThe data U2 has a size of: ');
fprintf('%d ', size(U2));
fprintf('\nDimension reduction for face dataset.\n\n');
K1 = 1;
[VR1,ASPE1,TV1] = varianceRetained(S1,K1);
while VR1 <= varRetained
    K1=K1+1;
    [VR1,ASPE1,TV1] = varianceRetained(S1,K1);
end
VRarray1 = zeros(1,K1);
Karray1 =  zeros(1,K1);
for i = 1:K1
    [VRarray1(i),~,~] = varianceRetained(S1,i);
    Karray1(i)=i;
end
fprintf(['Chandra K Principal Components = ',num2str(K1),'\n']);
fprintf(['Chandra Average Square Projection Error = ',num2str(ASPE1),'\n']);
fprintf(['Chandra Total Variation = ',num2str(TV1),'\n']);
fprintf(['Chandra Variance Retained = ',num2str(VR1*100),' %%\n']);
K2 = 1;
[VR2,ASPE2,TV2] = varianceRetained(S2,K2);
while VR2 <= varRetained
    K2=K2+1;
    [VR2,ASPE2,TV2] = varianceRetained(S2,K2);
end
VRarray2 = zeros(1,K2);
Karray2 =  zeros(1,K2);
for i = 1:K2
    [VRarray2(i),~,~] = varianceRetained(S2,i);
    Karray2(i)=i;
end
fprintf(['Zeki K Principal Components = ',num2str(K2),'\n']);
fprintf(['Zeki Average Square Projection Error = ',num2str(ASPE2),'\n']);
fprintf(['Zeki Total Variation = ',num2str(TV2),'\n']);
fprintf(['Zeki Variance Retained = ',num2str(VR2*100),' %%\n']);
figure(3);
subplot(1, 2, 1);
plot(Karray1, VRarray1);
title('Chandra Variance and Principal Component Graph');
ylabel('Variance');
xlabel('Principal Components');
subplot(1, 2, 2);
plot(Karray2, VRarray2);
```

```
title('Zeki Variance and Principal Component Graph');
ylabel('Variance');
xlabel('Principal Components');
Z1 = projectData(X_norm1, U1, K1);
Z2 = projectData(X_norm2, U2, K2);
fprintf('\nThe Chandra projected data Z has a size of: ')
fprintf('%d ', size(Z1));
fprintf('\nThe Zeki projected data Z has a size of: ')
fprintf('%d ', size(Z2));
fprintf('\n\nProgram paused. Press enter to continue.\n');
pause;
fprintf('\nVisualizing the projected (reduced dimension) images.\n\n');
X_rec1  = recoverData(Z1, U1, K1);
X_rec2  = recoverData(Z2, U2, K2);
fprintf('\nThe data X_rec1 has a size of: ')
fprintf('%d ', size(X_rec1));
fprintf('\nThe data X_rec2 has a size of: ')
fprintf('%d ', size(X_rec2));
figure(4);
subplot(2, 2, 1);
imshow(featureDenormalize(X_norm1, mu1, sigma1));
title('Orginal Chandra Image with Noise');
axis square;
subplot(2, 2, 2);
imshow(featureDenormalize(X_norm2, mu2, sigma2));
title('Orginal Zeki Image with Noise');
axis square;
subplot(2, 2, 3);
imshow(featureDenormalize(X_rec1, mu1, sigma1));
title(['Chandra Recovered Image with ',num2str(K1),' Principal Components ',10,' and
',num2str(VR1*100),'% Variance']);
axis square;
subplot(2, 2, 4);
imshow(featureDenormalize(X_rec2, mu2, sigma2));
title(['Zeki Recovered Image with ',num2str(K2),' Principal Components ',10,' and
',num2str(VR2*100),'% Variance']);
axis square;
```

APPENDIX B: FACIAL RECOGNITION PRINCIPAL COMPONENT ANALYSIS MATLAB CODE

```
function [X_norm, mu, sigma] = featureNormalize(X)
        mu = mean(X);
        X_norm = bsxfun(@minus, X, mu);
        sigma = std(X_norm);
        X_norm = bsxfun(@rdivide, X_norm, sigma);
end
function [U, S] = pca(X)
        [m,n] = size(X);
        U = zeros(n);
        S = zeros(n);
        Sigma = (1/m)*(X')*X;
        [U, S, ~] = svd(Sigma);
end
function [VR,ASPE,TV] = varianceRetained(S,K)
        [m,~] = size(S);
        ASPE = 0;
        TV = 0;
        for i = 1:m
            if i <= K
                ASPE = S(i,i) + ASPE;
            end
            TV = S(i,i) + TV;
        end
        VR = ASPE/TV;
end
function Z = projectData(X, U, K)
        Z = zeros(size(X, 1), K);
        Ureduce = U(:,1:K);
        Z = X*Ureduce;
end
function X_rec = recoverData(Z, U, K)
        X_rec = zeros(size(Z, 1), size(U, 1));
        Ureduce = U(:,1:K);
        X_rec = Z*Ureduce';
end
function [X] = featureDenormalize(X_norm, mu, sigma)
        X_sigma = bsxfun(@times, X_norm, sigma);
        X = bsxfun(@plus, X_sigma, mu);
end
function [h, display_array] = displayData(X, example_width)
        if ~exist('example_width', 'var') || isempty(example_width)
            example_width = round(sqrt(size(X, 2)));
        end
        colormap(gray);
         [m n] = size(X);
        example_height = (n / example_width);
        display_rows = floor(sqrt(m));
        display_cols = ceil(m / display_rows);
        pad = 1;
        display_array = - ones(pad + display_rows * (example_height + pad), ...
                               pad + display_cols * (example_width + pad));
        curr_ex = 1;
        for j = 1:display_rows
            for i = 1:display_cols
                if curr_ex > m,
                    break;
                end
                max_val = max(abs(X(curr_ex, :)));
                display_array(pad + (j - 1) * (example_height + pad) + (1:example_height),
        ...
                              pad + (i - 1) * (example_width + pad) + (1:example_width)) =
        ...
                                reshape(X(curr_ex, :), example_height, example_width) /
        max_val;
                curr_ex = curr_ex + 1;
            end
            if curr_ex > m,
                break;
            end
```

```
        end
        h = imagesc(display_array, [-1 1]);
        axis image off
        drawnow;
end

clc; clear; close all;
fprintf('Facial Recognition Example\n\n');
varRetained = 0.80;
gaussianNoiseMean = 0.02;
gaussianNoiseVariance = 0.005;
image_dims = [50, 50];
myFolder = '/Users/David/Desktop/CPP Faces';
filePattern = fullfile(myFolder, '*.jpg');
jpegFiles = dir(filePattern);
num_faces = numel(jpegFiles);
imageMatrix = zeros(num_faces, prod(image_dims));
noiseImageMatrix = zeros(num_faces, prod(image_dims));
for k = 1:length(jpegFiles)
    baseFileName = jpegFiles(k).name;
    fullFileName = fullfile(myFolder, baseFileName);
    fprintf(1, 'Now reading %s\n', fullFileName);
    imageArray = imread(fullFileName);
    imageArray = rgb2gray(imageArray);
    imageArray = im2double(imageArray);
    % Add noise to image
    noiseImageArray =
imnoise(imageArray,'gaussian',gaussianNoiseMean,gaussianNoiseVariance);
    imageArray = imresize(imageArray, image_dims);
    noiseImageArray = imresize(noiseImageArray, image_dims);
    imageMatrix(k,:) = double(imageArray(:).');
    noiseImageMatrix(k,:) = double(noiseImageArray(:).');
end
figure(1)
displayData(imageMatrix(1:(num_faces),:));
title('CPP Face Dataset')
fprintf('The data imageMatrix has a size of: ')
fprintf('%d ', size(imageMatrix));
figure(2)
displayData(noiseImageMatrix(1:(num_faces),:));
title('CPP Face Dataset with Noise')
fprintf('The data imageMatrix has a size of: ')
fprintf('%d ', size(noiseImageMatrix));
fprintf('\n\nProgram paused. Press enter to continue.\n');
pause;
[X_norm, mu, sigma] = featureNormalize(imageMatrix);
[X_norm_noise, mu_noise, sigma_noise] = featureNormalize(noiseImageMatrix);
figure(3)
displayData(X_norm(1:num_faces, :));
title('Normalize Faces Dataset')
figure(4)
displayData(X_norm_noise(1:num_faces, :));
title('Normalize Faces Dataset with Noise')
[U, S] = pca(X_norm);
fprintf('The data U has a size of: ')
fprintf('%d ', size(U));
[U_noise, S_noise] = pca(X_norm_noise);
fprintf('The data U with noise has a size of: ')
fprintf('%d ', size(U_noise));
figure(5)
displayData(U(:,1:(num_faces-1))');
title('Eigenfaces')
fprintf('\nDimension reduction for face dataset.\n\n');
figure(6)
displayData(U_noise(:,1:(num_faces-1))');
title('Eigenfaces with Noise')
fprintf('\nDimension reduction for face dataset with noise.\n\n');
K = 1;
[VR,ASPE,TV] = varianceRetained(S,K);
while VR <= varRetained
    K=K+1;
    [VR,ASPE,TV] = varianceRetained(S,K);
end
```

```
VRarray = zeros(1,K);
Karray =  zeros(1,K);
for i = 1:K
    [VRarray(i),~,~] = varianceRetained(S,i);
    Karray(i)=i;
end
fprintf(['K Principal Components = ',num2str(K),'\n']);
fprintf(['Average Square Projection Error = ',num2str(ASPE),'\n']);
fprintf(['Total Variation = ',num2str(TV),'\n']);
fprintf(['Variance Retained = ',num2str(VR*100),' %%\n']);
K_noise = 1;
[VR_noise,ASPE_noise,TV_noise] = varianceRetained(S_noise,K_noise);
while VR_noise <= varRetained
    K_noise=K_noise+1;
    [VR_noise,ASPE_noise,TV_noise] = varianceRetained(S_noise,K_noise);
end
VRarray_noise = zeros(1,K_noise);
Karray_noise =  zeros(1,K_noise);
for i = 1: K_noise
    [VRarray_noise(i),~,~] = varianceRetained(S_noise,i);
    Karray_noise(i)=i;
end
fprintf(['K Principal Components Noise = ',num2str(K_noise),'\n']);
fprintf(['Average Square Projection Error Noise = ',num2str(ASPE_noise),'\n']);
fprintf(['Total Variation Noise = ',num2str(TV_noise),'\n']);
fprintf(['Variance Retained Noise = ',num2str(VR_noise*100),' %%\n']);
figure(7);
plot(Karray, VRarray)
title('Variance and Principal Component Graph');
ylabel('Variance')
xlabel('Principal Components')
figure(8);
plot(Karray_noise, VRarray_noise)
title('Variance and Principal Component Graph with Noise');
ylabel('Variance')
xlabel('Principal Components')
Z = projectData(X_norm, U, K);
Z_noise = projectData(X_norm_noise, U_noise, K_noise);
fprintf('The projected data Z has a size of: ')
fprintf('%d ', size(Z));
fprintf('The projected data Z with noise has a size of: ')
fprintf('%d ', size(Z_noise));
fprintf('\n\nProgram paused. Press enter to continue.\n');
pause;
fprintf('\nVisualizing the projected (reduced dimension) faces.\n\n');
X_rec  = recoverData(Z, U, K);
X_rec_noise  = recoverData(Z_noise, U_noise, K_noise);
fprintf('The data X_rec has a size of: ')
fprintf('%d ', size(X_rec));
fprintf('The data X_rec has a size of: ')
fprintf('%d ', size(X_rec_noise));
figure(9)
subplot(1, 2, 1);
displayData(featureDenormalize(X_norm(1:num_faces,:), mu, sigma));
title('Original CPP Faces');
axis square;
subplot(1, 2, 2);
displayData(featureDenormalize(X_rec(1:num_faces,:), mu, sigma));
title(['Recovered Faces with ',num2str(K),' Principal Components ',10,' and
',num2str(VR*100),'% Variance']);
axis square;
figure(10)
subplot(1, 2, 1);
displayData(featureDenormalize(X_norm_noise(1:num_faces,:), mu_noise, sigma_noise));
title('Original CPP Faces with Noise');
axis square;
subplot(1, 2, 2);
displayData(featureDenormalize(X_rec_noise(1:num_faces,:), mu_noise, sigma_noise));
title(['Recovered Faces with ',num2str(K_noise),' Principal Components ',10,' and
',num2str(VR_noise*100),'% Variance with Noise']);
axis square;
```

APPENDIX C: KERNEL PRINCIPAL COMPONENT ANALYSIS PYTHON CODE

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import arange
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
def featureNormalize(X):
     rows, cols = X.shape
     mu = np.zeros((1, cols))
     sigma2 = np.zeros((1, cols))
     for i in range(0,cols):
          for j in range(0,rows):
               mu[0,i] = X[j,i] + mu[0,i]
          mu[0,i] = mu[0,i]/rows
     for i in range(0,cols):
          for j in range(0,rows):
               sigma2[0,i] = (X[j,i] - mu[0,i])**2 + sigma2[0,i]
          sigma2[0,i] = sigma2[0,i]/rows
     sigma = np.sqrt(sigma2)
     X_norm = (X-mu)/sigma
     return (X_norm, mu, sigma)
def rbf_kernel(X, gamma):
     # Calculating the squared Euclidean distances for every pair of points in the MxN
dimensional dataset.
     sq_dists = pdist(X, 'sqeuclidean')
     # Converting the pairwise distances into a symmetric MxM matrix.
     mat_sq_dists = squareform(sq_dists)
     # Computing the MxM kernel matrix.
     K = exp(-gamma * mat_sq_dists)
     return K
def recError(x_new,X, gamma, alphas, alphasKrow,sumalphas,Ksum):
     pair_dist = np.array([np.sum(np.power((x_new-row),2)) for row in X])
     K = np.exp(-gamma * pair_dist)
     N = K.shape[0]
     pair_dist_new = np.array([np.sum(np.power((x_new-x_new),2))])
     Knew = np.exp(-gamma * pair_dist_new)
     f = K.dot(alphas) - alphasKrow - sumalphas*(np.sum(K,axis=0)/N) + sumalphas*Ksum
     error = Knew - 2*(np.sum(K,axis=0)/N) + Ksum - f*f.T
     return error
def rbf_kpca(X,gamma,n_components):
     K = rbf_kernel(X,gamma)
     # Centering the symmetric NxN kernel matrix.
     N = K.shape[0]
     # Correct K for non-zero center of data in feature space
     Krow = np.sum(K,axis=0)/N
     Ksum = np.sum(Krow)/N
     for i in range(N):
          for j in range(N):
               K[i,j] = K[i,j] - Krow[i] - Krow[j] + Ksum
     # Obtaining eigenvalues in descending order with corresponding eigenvectors from the
symmetric matrix.
     eigvals, eigvecs = np.linalg.eigh(K)
     # Obtaining the i eigenvectors (alphas) that corresponds to the i highest
eigenvalues (lambdas).
     alphas = np.column_stack((eigvecs[:,-i] for i in range(1,n_components+1)))
     lambdas = np.array([eigvals[-i] for i in range(1,n_components+1)])
     lambdas = np.diag(lambdas)
     # Residual Variance
     resvar = (np.trace(K)-np.trace(lambdas))/np.trace(K)
     print("Residual Variance: {}%\n".format(100*resvar))
     # Normalized Eigenvectors
     alphas = np.mat(alphas) * np.linalg.inv(np.sqrt(lambdas));
     sumalphas = np.sum(alphas,axis=0)
     alphasKrow = np.mat(Krow) * np.mat(alphas)
     # Evaluating reconstruction error for all data points
     error = np.zeros((N, 1))
     for i in range(N):
          x_new = X[i,:]
          error[i] = recError(x_new, X, gamma, alphas, alphasKrow,sumalphas,Ksum)
```

```
# Find max reconstruction error
     maxerror = np.max(error)
     print("Max Error: {}\n".format(maxerror))
     # Computing reconstruction error over data space
     xymin = np.min(X,axis=0)
     xymax = np.max(X,axis=0)
     dataRange = xymax-xymin
     # Provide some space around data points
     offset = dataRange*0.05
     xymin = xymin-offset
     xymax = xymax+offset
     m = 100 # choose even value
     xx = np.linspace(xymin[0], xymax[0], m)
     yy = np.linspace(xymin[1], xymax[1], m)
     x1,x2 = np.meshgrid(xx,yy,sparse=False, indexing='ij')
     x1=x1.T
     x2=x2.T
     err = np.zeros((m,m))
     for i in range(m):
          for j in range(m):
               x_new = np.array([x1[0,j], x2[i,0]])
               err[i,j] = recError(x_new, X, gamma, alphas, alphasKrow,sumalphas,Ksum)
     return alphas, lambdas, K, maxerror, err, x1, x2

dataCol = 6
windowSize = 4000
window = 0
startSamples = windowSize*window
endSamples = windowSize*(window + 1)
X_all = np.loadtxt('sat_data_recorder_nominal_training_set_desampled.csv',delimiter=',')
X_norm,_,_ = featureNormalize(X_all)
Xnom = np.column_stack((X_all[startSamples:endSamples, 0],
X_norm[startSamples:endSamples, dataCol]))
Xoffnom_all = np.loadtxt('sat_data_recorder_off_nominal_desampled.csv',delimiter=',')
Xoffnom_norm,_,_ = featureNormalize(Xoffnom_all)
Xoffnom = np.column_stack((Xoffnom_all[startSamples:endSamples, 0],
Xoffnom_norm[startSamples:endSamples, dataCol]))
X = np.matrix(Xoffnom[:windowSize, 0]).T
y = np.matrix(Xoffnom[:windowSize, 1]).T
plt.figure(1)
plt.plot(Xnom[:, 0],Xnom[:, 1],marker='.',color='b',linestyle='none')
plt.xlabel('Time')
plt.ylabel('Column {}'.format(dataCol))
plt.title('Nominal Data')
plt.show()
plt.figure(2)
plt.plot(Xoffnom[:, 0],Xoffnom[:, 1],marker='.',color='b',linestyle='none')
plt.xlabel('Time')
plt.ylabel('Column {}'.format(dataCol))
plt.title('Off Nominal Data')
plt.show()
gamma = 0.0005
n_components = 250
alphas, lambdas, K, maxerror, err, x1, x2 = rbf_kpca(Xnom, gamma, n_components)
fig = plt.figure(3)
ax = fig.add_subplot(212)
ax.plot(Xoffnom[:, 0],Xoffnom[:, 1])
levels = [maxerror*.1,maxerror*0.5,maxerror]
cs = ax.contour(x1,x2,err,levels,color='r', linestyles='solid')
ax.set_title('Off Nominal Data (gamma = {}, pc = {})'.format(gamma,n_components))
ax.set_xlabel('Time')
plt.ylabel('Column {}'.format(dataCol))
plt.clabel(cs, inline = True)
ax = fig.add_subplot(211)
ax.plot(Xnom[:, 0],Xnom[:, 1])
levels = [maxerror*.1,maxerror*0.5,maxerror]
cs = ax.contour(x1,x2,err,levels,color='r', linestyles='solid')
ax.set_title('Nominal Data (gamma = {}, pc = {})'.format(gamma,n_components))
ax.set_xlabel('Time')
plt.ylabel('Column {}'.format(dataCol))
plt.clabel(cs, inline = True)
plt.show()
```

APPENDIX D: K-MEANS CLUSTERING PYTHON CODE

```
from time import time
import numpy as np
from sklearn.cluster import KMeans
from sklearn.preprocessing import scale
import matplotlib.pyplot as plt
def sliding_chunker(data, window_len, slide_len):
     chunks = []
     for pos in range(0, int(len(data)), int(slide_len)):
          chunk = np.copy(data[pos:pos+window_len])
          if len(chunk) != window_len:
               continue
          chunks.append(chunk)
     return chunks
def get_windowed_segments(data, window):
     step = 2
     windowed_segments = []
     segments = sliding_chunker(
          data,
          window_len=len(window),
          slide_len=step
     )
     print("Produced %d waveform segments" % len(segments))
     for segment in segments:
          windowed_segment = np.copy(segment) * window
          windowed_segments.append(windowed_segment)
     return windowed_segments
def reconstruct(data, window, clusterer):
     window_len = len(window)
     slide_len = window_len/2
     segments = sliding_chunker(data, window_len, slide_len)
     windowed_segments = []
     for segment in segments:
          windowed_segment = np.copy(segment) * window
          windowed_segments.append(windowed_segment)
     windowed_segments = np.matrix(windowed_segments)
     reconstructed_data = np.zeros(len(data))
     for segment_n, segment in enumerate(windowed_segments):
          nearest_match_idx = clusterer.predict(segment)[0]
          nearest_match = np.copy(clusterer.cluster_centers_[nearest_match_idx])
          pos = segment_n * slide_len
          reconstructed_data[pos:pos+window_len] += nearest_match
     return reconstructed_data
def plot_waves(waves, step):
     plt.figure()
     n_graph_rows = 4
     n_graph_cols = 3
     graph_n = 1
     wave_n = 0
     for _ in range(n_graph_rows):
          for _ in range(n_graph_cols):
               axes = plt.subplot(n_graph_rows, n_graph_cols, graph_n)
               plt.plot(waves[wave_n])
               graph_n += 1
               wave_n += step
     # fix subplot sizes so that everything fits
     plt.tight_layout()
     plt.show()

dataCol = 6
windowSize = 20000
WINDOW_LEN = 32
window = 0
K = 1200
percentile = 99
startSamples = windowSize*window
endSamples = windowSize*(window + 1)
X_all = np.loadtxt('sat_data_recorder_nominal_training_set_desampled.csv',delimiter=',')
Xnom = np.column_stack((X_all[startSamples:endSamples, 0], X_all[startSamples:endSamples,
dataCol]))
Xoffnom_all = np.loadtxt('sat_data_recorder_off_nominal_desampled.csv',delimiter=',')
```

```
Xoffnom = np.column_stack((Xoffnom_all[startSamples:endSamples, 0],
Xoffnom_all[startSamples:endSamples, dataCol]))
Xnom = Xnom[:windowSize, 1].T
Xoffnom = Xoffnom[:windowSize, 1].T
Xnom = scale(Xnom[:windowSize, 1].T)
Xoffnom = scale(Xoffnom[:windowSize, 1].T)
plt.figure(1)
plt.plot(Xnom[0:windowSize])
plt.xlabel("Sample number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Nominal Data')
plt.show()
plt.figure(2)
plt.plot(Xoffnom[0:windowSize])
plt.xlabel("Sample number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Off Nominal Data')
plt.show()
plt.figure(3)
plt.subplot(211)
plt.plot(Xnom[0:windowSize])
plt.xlabel("Sample number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Nominal Data')
plt.subplot(212)
plt.plot(Xoffnom[0:windowSize])
plt.xlabel("Sample number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Off Nominal Data')
plt.show()
n_plot_samples = windowSize
window_rads = np.linspace(0, np.pi, WINDOW_LEN)
window = np.sin(window_rads)**2
plt.plot(window)
plt.show()
print("Windowing data...")
windowed_segments = get_windowed_segments(Xnom, window)
windowed_segments = np.matrix(windowed_segments)
plot_waves(windowed_segments, step=2)
print("Clustering...")
clusterer = KMeans(n_clusters=K)
clusterer.fit(windowed_segments)
print("Reconstructing...")
reconstruction = reconstruct(Xnom, window, clusterer)
error = np.abs(reconstruction - Xnom)
error_99th_percentile = np.percentile(error[0:n_plot_samples], percentile)
maxError = np.max(error[0:n_plot_samples])
meanError = np.mean(error[0:n_plot_samples])
new_reconstruction = reconstruct(Xoffnom[0:n_plot_samples], window, clusterer)
new_error = np.abs(new_reconstruction[0:n_plot_samples] - Xoffnom[0:n_plot_samples])
new_maxError = np.max(new_error[0:n_plot_samples])
new_meanError = np.mean(new_error[0:n_plot_samples])
new_error_99th_percentile = np.percentile(new_error[0:n_plot_samples], percentile)
anomaly_counter = 0
threshold = np.percentile(new_error[0:n_plot_samples], 90)
threshold_line = np.zeros(len(new_error))
threshold_line[0:len(new_error)] = threshold
thershold_percent = 0.65
window_len = 400
slide_len = window_len/10
segments = sliding_chunker(new_error, window_len, slide_len)
anomaly_detected = np.zeros(len(new_error))
for segment_n, segment in enumerate(segments):
     segment = np.copy(segment)
     for segment_index in segment:
          if segment[segment_index] >= threshold:
               anomaly_counter = anomaly_counter + 1
     pos = segment_n * slide_len
     if anomaly_counter >= window_len*thershold_percent:
          anomaly_detected[pos:pos+window_len] = 300
     else:
          anomaly_detected[pos:pos+window_len] = 0
     anomaly_counter = 0
```

```
plt.figure(4)
plt.plot(Xoffnom[0:n_plot_samples], label="Original Off Nominal Data")
plt.plot(new_reconstruction[0:n_plot_samples], label="Reconstructed Off Nominal Data")
plt.plot(new_error, label="Reconstruction error")
plt.plot(anomaly_detected, label="Anomaly Detected")
plt.plot(threshold_line, label="Threshold = %.1f" % threshold)
plt.xlabel("Sample Number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Reconstructed Off Nominal Data with K = %d Clusters' % K)
plt.xlim([0, n_plot_samples])
plt.ylim([0, 600])
plt.legend()
plt.show()
plt.figure(5)
plt.plot(Xnom[0:n_plot_samples], label="Original Nominal Data")
plt.plot(reconstruction[0:n_plot_samples], label="Reconstructed Nominal Data")
plt.plot(error[0:n_plot_samples], label="Reconstruction error")
plt.xlabel("Sample Number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Reconstructed Nominal Data with K = %d Clusters' % K)
plt.xlim([0, n_plot_samples])
plt.ylim([0, 400])
plt.legend()
plt.show()
print("Maximum reconstruction error is %.1f" % maxError)
print("Reconstruction error mean is %.1f" % meanError)
print("99th percentile of reconstruction error was %.1f" % error_99th_percentile)
plt.figure(6)
plt.plot(Xoffnom[0:n_plot_samples], label="Original Off Nominal Data")
plt.plot(new_reconstruction[0:n_plot_samples], label="Reconstructed Off Nominal Data")
plt.plot(new_error[0:n_plot_samples], label="Reconstruction Error")
plt.xlabel("Sample Number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Reconstructed Off Nominal Data with K = %d Clusters' % K)
plt.xlim([0, n_plot_samples])
plt.ylim([0, 400])
plt.legend()
plt.show()
print("Maximum reconstruction error was %.1f" % new_maxError)
print("Reconstruction error mean is %.1f" % new_meanError)
print("99th percentile of reconstruction error was %.1f" % new_error_99th_percentile)
plt.figure(7)
plt.subplot(211)
plt.plot(Xnom[0:n_plot_samples], label="Original Nominal Data")
plt.plot(reconstruction[0:n_plot_samples], label="Reconstructed Nominal Data")
plt.plot(error[0:n_plot_samples], label="Reconstruction error")
plt.xlabel("Sample Number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Reconstructed Nominal Data with K = %d Clusters' % K)
plt.xlim([0, n_plot_samples])
plt.ylim([0, 400])
plt.legend()
plt.subplot(212)
plt.plot(Xoffnom[0:n_plot_samples], label="Original Off Nominal Data")
plt.plot(new_reconstruction[0:n_plot_samples], label="Reconstructed Off Nominal Data")
plt.plot(new_error[0:n_plot_samples], label="Reconstruction Error")
plt.xlabel("Sample Number")
plt.ylabel('Column {}'.format(dataCol))
plt.title('Reconstructed Off Nominal Data with K = %d Clusters' % K)
plt.xlim([0, n_plot_samples])
plt.ylim([0, 400])
plt.legend()
plt.show()
```

APPENDIX E: RECURRENT NEURAL NETWORKS PYTHON CODE

```
import matplotlib.pyplot as plt
import numpy as np
import math
import theano as theano
import theano.tensor as T
import operator
import copy
import time
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from keras.layers.core import Dense, Activation, Dropout
from keras.layers.recurrent import LSTM
from keras.models import Sequential
from keras.models import model_from_json
# fix random seed for reproducibility
np.random.seed(7)
data_column = 6
sequence_length = 50
def dropin(X, y):
     X_hat = []
     y_hat = []
     for i in range(0, len(X)):
          for j in range(0, np.random.random_integers(0,20)):
               X_hat.append(X[i, :])
               y_hat.append(y[i])
     return np.asarray(X_hat), np.asarray(y_hat)
def get_split_prep_data(train_start, train_end,test_start, test_end):
     # Generate Sample Data
     nominal_data_file_name = 'sat_data_recorder_nominal_training_set_desampled.csv'
     off_nominal_data_file_name = 'sat_data_recorder_off_nominal_desampled.csv'
     windowSize = 20000
     window = 0
     startSamples = windowSize*window
     endSamples = windowSize*(window + 1)
     nominal_data = np.loadtxt(nominal_data_file_name,delimiter=',')
     nominal_data = np.column_stack((nominal_data[startSamples:endSamples, 0],
nominal_data[startSamples:endSamples,      data_column]))
     off_nominal_data = np.loadtxt(off_nominal_data_file_name,delimiter=',')
     off_nominal_data = np.column_stack((off_nominal_data[startSamples:endSamples, 0],
off_nominal_data[startSamples:endSamples, data_column]))
     nominal_data = nominal_data[:windowSize, 1].T
     off_nominal_data = off_nominal_data[:windowSize, 1].T
     # Normalize the Dataset
     scaler = MinMaxScaler(feature_range=(0, 1))
     nominal_data = scaler.fit_transform(nominal_data)
     off_nominal_data= scaler.fit_transform(off_nominal_data)
     # Train Data
     print("Creating Train Data...")
     result = []
     for index in range(train_start, train_end - sequence_length):
          result.append(nominal_data[index: index + sequence_length])
     result = np.array(result)
     result, result_mean = z_norm(result)
     train = result[train_start:train_end, :]
     np.random.shuffle(train)
     X_train = train[:, :-1]
     y_train = train[:, -1]
     X_train, y_train = dropin(X_train, y_train)
     # Test Data
     print("Creating Test Data...")
     result = []
     for index in range(test_start, test_end - sequence_length):
          result.append(off_nominal_data[index: index + sequence_length])
     result = np.array(result)
     result, result_mean = z_norm(result)
     X_test = result[:, :-1]
     y_test = result[:, -1]
     X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
     X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
     return X_train, y_train, X_test, y_test, nominal_data, off_nominal_data
```

```
def build_model():
     model = Sequential()
     layers = {'input': 1, 'hidden1': 128, 'hidden2': 256 , 'output': 1}
     model.add(LSTM(
          input_length=sequence_length - 1,
          input_dim=layers['input'],
          output_dim=layers['hidden1'],
          return_sequences=True))
     model.add(Dropout(0.2))
     model.add(LSTM(
          layers['hidden2'],
          return_sequences=False))
     model.add(Dropout(0.2))
     model.add(Dense(
          output_dim=layers['output']))
     model.add(Activation("linear"))
     start = time.time()
     model.compile(loss="mse", optimizer="rmsprop")
     print("Compilation Time : ", time.time() - start)
     return model
def save_model(model=None):
     # serialize model to JSON
     model_json = model.to_json()
     with open("model.json", "w") as json_file:
          json_file.write(model_json)
     # serialize weights to HDF5
     model.save_weights("model.h5")
     print("Saved model to disk")
def run_network(model=None, data=None):
     global_start_time = time.time()
     epochs = 300
     if data is None:
          print('Loading data... ')
          X_train, y_train, X_test, y_test, nominal_data, off_nominal_data =
get_split_prep_data(
                0, 3000, 0, 3000)
     else:
          X_train, y_train, X_test, y_test = data
     print('\nData Loaded. Compiling...\n')
     if model is None:
          model = build_model()
     try:
          print("Training")
          model.fit(
               X_train, y_train,
               batch_size=512, nb_epoch=epochs, validation_split=0.05)
          print("Predicting")
          predicted = model.predict(X_test)
          print("shape of predicted", np.shape(predicted), "size", predicted.size)
          print("Reshaping predicted")
          predicted = np.reshape(predicted, (predicted.size,))
     except KeyboardInterrupt:
          print("prediction exception")
          print('Training duration (s) : ', time.time() - global_start_time)
          return model, y_test, 0
     print('Training duration (s) : ', time.time() - global_start_time)
     return model, X_test, y_test, predicted, nominal_data, off_nominal_data

model, X_test, y_test, predicted, nominal_data, off_nominal_data= run_network()
save_model(model)
import h5py
with h5py.File('model.h5','r') as hf:
     print('List of arrays in this file: \n', hf.keys())
# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")
loaded_model.compile(loss="mse", optimizer="rmsprop")
predicted = loaded_model.predict(X_test)
```

```
print("Reshaping predicted")
predicted = np.reshape(predicted, (predicted.size,))
plt.figure(1)
plt.subplot(211)
plt.plot(nominal_data)
plt.xlabel("Sample Number")
plt.ylabel('Column {}'.format(data_column))
plt.title('Nominal Data')
plt.subplot(212)
plt.plot(off_nominal_data)
plt.xlabel("Sample Number")
plt.ylabel('Column {}'.format(data_column))
plt.title('Off Nominal Data')
plt.show()
plt.figure(2)
plt.subplot(311)
plt.title("Signal with Anomalies")
plt.ylabel('Column {}'.format(data_column))
plt.plot(y_test[:len(y_test)], 'b')
plt.subplot(312)
plt.title("Predicted Signal")
plt.ylabel('Column {}'.format(data_column))
plt.plot(predicted[:len(y_test)], 'g')
plt.subplot(313)
plt.title("Squared Error")
plt.xlabel("Sample Number")
plt.ylabel('Column {}'.format(data_column))
mse = ((y_test - predicted) ** 2)
plt.plot(mse, 'r')
plt.show()
```