Nonlinear Dimension Estimation for Cryptocurrency Data [1]

Lyra Layne, Cole Peterson,
Benjamin Rajotte, Taenler Tavares

Advisors: Fangfang Wang and Stephan Sturm

A Major Qualifying Project Submitted to the Faculty of WORCESTER POLYTECHNIC INSTITUTE in partial fulfillment of the requirements for the Degrees of Bachelor of Science in Mathematics and Bachelor of Science in Data Science.

Department of Mathematical Sciences
Worcester Polytechnic Institute
Worcester, MA

[1] This report represents the work of four WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.

Acknowledgements

We extend our heartfelt thanks to everyone who supported us throughout this project. To our advisors, Professor Stephan Sturm and Professor Fangfang Wang, thank you for keeping us moving in the right direction and for your invaluable feedback and advice. To Aaron Stapleton, thank you for generously taking the time to discuss with us the report and code we based this project on.

Contents

Chapter 1

Introduction

Since the advent of blockchain technology in 2009 and the creation of the first major cryptocurrency Bitcoin, the cryptocurrency market has become increasingly prevalent. Presently, tens of thousands of cryptocurrencies exist with a total market capitalization of over $1.1T. With cryptocurrency prices rising and falling constantly, this market behaves somewhat like the stock market.

In the stock market, investing in a portfolio of diverse stocks can often lower an investor's overall risk; specifically, by investing in stocks with low correlations to each other, investors can minimize their losses in the case of markets crashing or companies going under. By investigating the correlations between various cryptocurrencies, we will determine the nature of diversity within the cryptocurrency market to understand how a cryptocurrency investment portfolio can be diversified.

To analyze how the values of cryptocurrencies correlate with one another, we will use various methods of dimension reduction such as principal component analysis (PCA), robust principal component analysis (RPCA), and autoencoders. These methods will not only provide representations of market statistics with smaller datasets but also show how much of the variance in value of each cryptocurrency is represented by the components which make up these datasets.

This paper was motivated by a 2021 REU project done by four undergraduate students at Worcester Polytechnic Institute, entitled *Decorrelation Detection in a Financial Time Series Data Set* [9]. The project considered a dataset of exchange traded funds provided by their sponsor, Wellington Management, and compared the robustness of dimension estimation methods, which they used to visualize interdependencies between the assets. Their work outlined the methodology used in this project.

The rest of this paper is organized as follows: we begin in Chapter 2 by providing background information on the cryptocurrency market, the analyzed data, and the methods we use to analyze said data. Chapter 3 then explains our process of estimating the target output dimension of these methods using various dimension estimation methods on an expanding window scheme (EWS). Chapter 4 displays and explains the results from these methods, and Chapter 5 subsequently analyzes these results and further researches the correspondence between notable dates and anomalies in the market. Finally, in Chapter 6 we draw conclusions from these analyses about the dimensionality of the cryptocurrency market and the nature of a diverse cryptocurrency investment portfolio.

Chapter 2

Background

In this chapter, the necessary background is given to understand the methods and results presented in this paper. A brief introduction to cryptocurrencies is provided, followed by a description of the data set we used. The mathematical theory behind the three methods we considered is then given. Some additional background for readers not familiar with matrix decompositions, machine learning and neural networks is provided in the appendices.

2.1 Data

2.1.1 Cryptocurrencies

The Cryptocurrency market has experienced significant growth and volatility in recent years. From August 18th, 2020, to January 24th, 2023, the cryptocurrency market has seen both ups and downs. Bitcoin, the most popular cryptocurrency with the highest market capitalization, has experienced a series of price swings during this period. In particular, Bitcoin went through a bullish run in late 2020 and early 2021, reaching an all-time high of $64,863.10 on April 14th, 2021, before experiencing a significant price correction and falling to around $18,000 [8].

One of the significant factors driving the cryptocurrency market is the increasing adoption of blockchain technology. Blockchain technology is a decentralized and distributed ledger that enables secure and transparent transactions without the need for intermediaries [17]. The adoption of blockchain technology has the potential to disrupt various industries, such as finance, healthcare, and supply chain management.

Another factor contributing to the growth of the cryptocurrency market is the increasing interest from institutional investors such as Deutsche Bank, Wells Fargo, Citigroup, and Bank of America, who have been attracted to the cryptocurrency market due to its potential for high returns. Moreover, the increasing number of cryptocurrency exchanges and investment vehicles has made it easier for institutional investors to access the cryptocurrency market. However, the cryptocurrency market has also been subject to regulatory scrutiny and security risks. Regulatory uncertainty and security risks have been cited as significant barriers to the widespread adoption of cryptocurrencies [23]. As such, researchers have been investigating the potential impact of regulations on the cryptocurrency market.

Rank	Cryptocurrency	Ticker	Year Established
1	Bitcoin	BTC	2009
2	Ethereum	ETH	2014
3	Binance Coin	BNB	2017
4	Binance USD	BUSD	2019
5	Ripple	XRP	2012
6	Cardona	ADA	2015
7	Dogecoin	DOGE	2013
8	Polygon	MATIC	2012
9	Solana	SOL	2020
10	Polkadot	DOT	2020
11	Tron	TRX	2017
12	LiteCoin	LTC	2011
13	ChainLink	LINK	2017

Table 2.1: Analyzed cryptocurrencies

The coins listed in Figure 2.1 were selected based on the availability of data from Binance, market capitalization, and the year in which they were established. When choosing assets to represent the cryptocurrency market, we look at the total market capitalization of assets; the list is the 13 highest market cap cryptocurrencies which make up over 90% of the market capitalization. Market capitalization measures the respective market values of the selected assets by multiplying the price of one unit by the total units available. Utilizing these 13 cryptocurrencies that make up most of the market capitalization results in well-represented market data. Using the closing prices P_t of each day for all 13 assets, we calculate daily log returns, given by $R_t = \ln(P_t/P_{t-1})$. Data for this analysis are provided by a Binance API and are comprised of the daily closing prices on 13 crytocrrencies from 08/18/2022 to 01/24/2023, a total of 890 trading days.

2.1.2 Binance API

We access a Binance API in Python to extract the necessary historical data managed by Binance, a leading cryptocurrency exchange. This historical information is completely public and open source for educational purposes. An API, or application programming interface, acts as a communication layer that allows different systems to communicate without understanding precisely what the other does. The API takes access data and integrates them into the Python library so they can be analyzed and processed within Juypter Notebook. Each individual ticker will have its own CSV file which is written out in the API. When using R and Python, these CSV files can be read into a data set by ticker, pooling all data together to then be analyzed.

All of the Python Libraries used to access the Binance API are listed in Appendix B, as well as the API base URL and necessary dependencies to access the data and permissions.

The historical data are accessed for each ticker of the 13 cryptocurrencies selected. Each datum contains the ticker, the desired time series, and the amount of time, which for our case was daily. The data are then constructed into a dataset with price, volume, and time included. These data are then written into a CSV file, which can then be read into datasets in other systems as needed.

2.1.3 Exploratory Data Analysis

Rank	Ticker	Mean	Median	St. Dev.	Kurtosis	Skewness
1	BTC	7.18E-04	0.000407	0.036914	2.78071	-0.244131
2	ETH	1.46E-03	0.00317	0.050008	3.912489	-0.424909
3	BNB	2.89E-03	0.001922	0.056564	14.903465	0.621005
4	BUSD	7.87E-07	0	0.000421	31.669809	1.799455
5	XRP	3.28E-04	0.001196	0.065936	12.901975	0.152958
6	ADA	1.07E-03	0.000614	0.058164	3.071133	0.241235
7	DOGE	3.58E-03	-0.000453	0.094965	95.211645	6.045469
8	MATIC	2.18E-03	0.001244	0.078254	6.26063	0.97253
9	SOL	4.02E-03	-0.000022	0.078418	5.891495	-0.30449
10	DOT	7.73E-04	0.000238	0.065877	7.688474	0.222919
11	TRX	8.38E-04	0.00207	0.052318	7.961459	-0.108769
12	LTC	3.22E-04	0.001714	0.05468	7.393962	-0.84033
13	LINK	-9.99E-04	0.003372	0.064057	4.658368	-0.552381

Table 2.2: Summary statistics for the 13 cryptocurrecies

Exploratory data analysis is an important part of statistical analysis; exploring the data and observing big changes in fundamental statistics can motivate us to find and solve new problems. We first want to examine how each asset in particular changes over time. We observe our data as a whole in Table 2.2. The first column lists the ranks and the second column lists the tickers for the 13 assets. The last five columns show important summary statistics including mean, median, standard deviation, skewness, and kurtosis. The asset return mean of the greatest magnitude is SOL with 4.02×10^{-3}, with the rest being closer to 0, indicating the asset returns are very weakly serially correlated or near white noise. Further, a one-sample t-test indicates that all the mean returns are insignificantly different from zero.

Standard deviation is a statistical measure of the amount of variability or dispersion in a set of data. It tells us how spread out the data is from the mean or average value. In the context of cryptocurrencies, standard deviation is often used to measure the volatility of prices. A larger standard deviation indicates a higher degree of volatility or risk in price movements, while a smaller standard deviation indicates the opposite. DOGE we can see has the highest standard deviation with .094 and the lowest was BUSD with .000421.

Kurtosis quantifies how much of the total probability lies in the tails of a distribution. A standard normal distribution has a kurtosis value of 3. A kurtosis value greater than 3, or excess kurtosis, indicates that the distribution has heavier tails than those of a normal distribution, implying that the distribution has a higher probability of extreme events or outliers; this is because the tails of the distribution contain more of the total probability. Such a distribution is called leptokurtic. On the other hand, a kurtosis value less than 3 indicates that the distribution has lighter tails than those of a normal distribution, implying that the distribution has a lower probability of extreme events or outliers. Of the 13 cryptocurrencies analyzed, all except BTC are leptokurtic.

In statistics, skewness measures the degree of asymmetry in a probability distribution. A normal distribution is symmetric, meaning that the left and right tails are identical in shape and size, and has a skewness of 0. In the context of cryptocurrencies, skewness can provide insight into the distribution of price returns over time. Of the 13 cryptocurrencies analyzed, 6 of their returns—BTC, ETH, SOL, TRX, LTC, and LINK—are negatively skewed, and all the other cryptocurrencies are positively skewed. Positive skewness indicates that the distribution has a long right tail, meaning that extreme positive returns occur more frequently than extreme negative returns. This suggests a greater potential for positive returns in the future. Conversely, a negative skewness indicates that the distribution has a long left tail, meaning that extreme negative returns occur more frequently than extremely positive returns. This suggests a greater risk of large losses in the future. Studies have shown that many cryptocurrencies have highly skewed return distributions, with a tendency towards large positive returns but also significant risks of large losses [16].

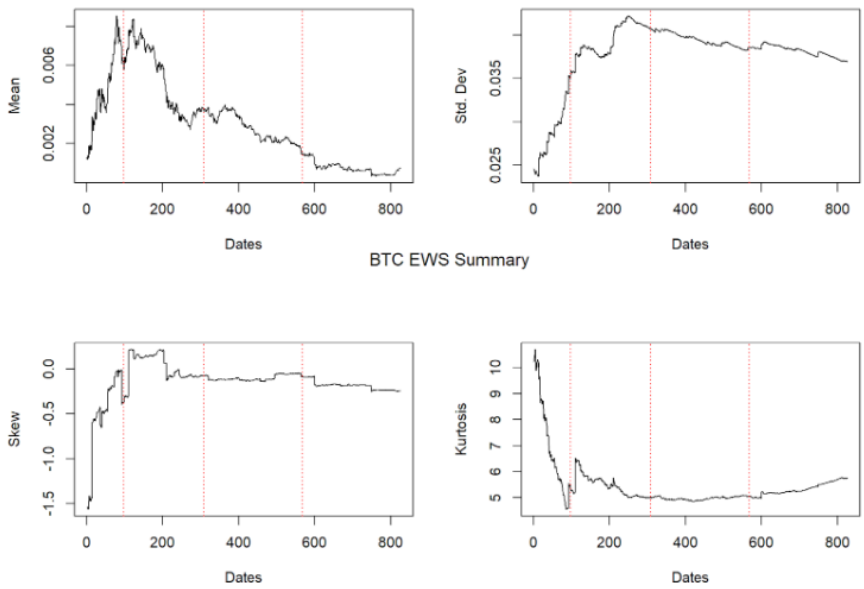


Figure 2.1: Summary statistics for Bitcoin across the EWS

The plots of summary statistics for all assets are available in Appendix C and Figure 2.1. In general, these plots show the means of returns are about zero, and many of them experience a sudden change on the three dates indicated by vertical red dotted lines. All of the assets' returns show sudden increases in standard deviation at the three points in time. Finally, almost every returns series have high volatility in the skewness and high volatility in excess kurtosis which occurs suddenly before or after each date. These sudden changes over time require that we use robust techniques for dimension estimation and data reconstruction that are resistant to outliers.

The heatmap shown in Figure 2.2 visualizes the correlations across the 13 asset returns. ETH and BTC were the most correlated with a correlation coefficient of .81. In addition ETH and LTC also share a value of .81 making the pairs the most correlated of the set. BUSD is the least correlated to the other 12 cryptocurrencies, having a negative correlation coefficient with all of them. This lack of correlation can be explained by its unique behavior as a stablecoin—a coin designed to retain a certain monetary value. DOGE is the least uncorrelated to the rest of the cryptocurrencies and is also the most volatile asset.

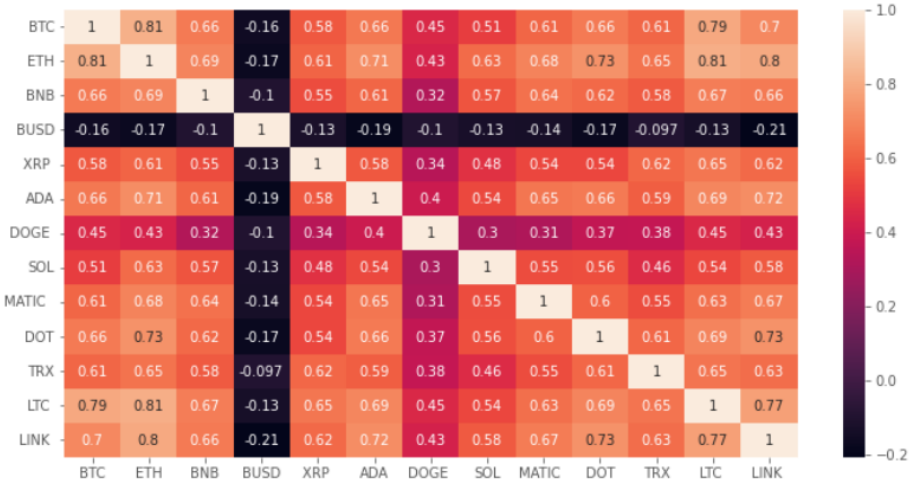


Figure 2.2: Heatmap of correlations between the 13 asset returns

2.2 Expanding Window Scheme

An expanding window is an iterative method for comparing data metrics as more data is introduced. We start with an arbitrary fixed starting point, $i = 64$, then incorporate more data at each iteration. For our data, we start with initial data matrix $A^{(0)}$ which has 64 data points corresponding to daily log-returns from August 19, 2020 to October 20, 2020 for our 13 assets. Each asset vector is of the form $A_{i,j} \in \mathbb{R}^{i\times 1}$ where $1 \leq j \leq 13$. Then, at each iteration for $k \leq 826$, we update $A^{(k)}$ such that $A^{(k)} = (A_{i+k,1}, \ldots, A_{i+k,13})$ and perform analysis on the new data matrix until we have our original data set.

We opted to use an expanding window scheme (EWS) as opposed to a rolling window scheme (RWS), which is similar in concept except the starting point is not fixed and changes with the upper bound. This choice was made so that we can visualize the performance of these methods on all of the data rather than a fixed amount of data. Furthermore, the dimension reduction methods require large data sets, which means a large window is needed for a RWS, and given the relatively small size of our data set, we felt an EWS would produce more interpretable results.

2.3 Principal Component Analysis

Principal component analysis (PCA) is a dimension reduction method with applications to genetics, facial recognition, finance, and more [24]. When given a data set, PCA forms a new basis of orthogonal vectors that successively maximize variance. PCA is used to analyze large data sets with many dimensions or variables by preserving as much statistical information as possible, while also increasing interpretabilty [12]. Mathematically, this means finding a new set of variables, or principal components, that are linear combinations of the original data set, which maximize variance and are uncorrelated to each other.

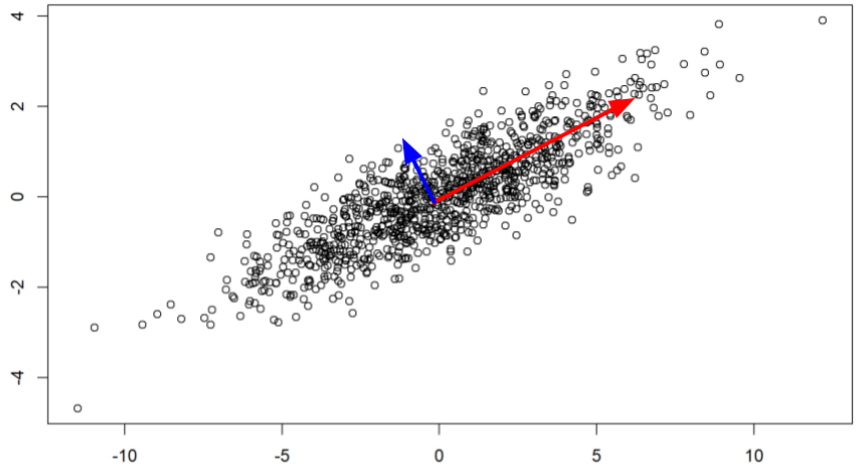


Figure 2.3: A multivariate Gaussian distribution with two orthogonal vectors corresponding to the directions of the greatest variance in the data.

In order to find these principal components, suppose we have a random vector $X \in \mathbb{R}^n$. The first principal component (PC) is the unit vector $u_1 \in \mathbb{R}^m$ which maximizes the variance of $u_1^\top X$. The next PC vector u_2 maximizes the variance of $u_2^\top X$ while being orthogonal to u_1. For $k \leq n$, the k-th PC vector is the vector that maximizes $u_k^\top X$ while being orthogonal to the previous $k-1$ PC vectors.

The variance of $u_k^\top X$ can be shown to be $\text{var}(u_k^\top X) = u_k^\top \hat{\Sigma} u_k$, where $\hat{\Sigma}$ is the covariance matrix of X. Then the vector which maximizes the variance also maximizes the convex quadratic, $u_k^\top \hat{\Sigma} u_k$. This is equivalent to maximizing

the following Lagrangian function:

$$\mathcal{L}(u, \lambda, \gamma_1, ..., \gamma_{k-1}) = u_k^\top \hat{\Sigma} u_k - \lambda(u_k^\top u_k - 1) - \sum_{i=1}^{k-1} 2\gamma_i u_i^\top u.$$

Differentiating with respect to u_k and equating $\mathcal{L}(*)$ to 0, we get

$$0 = \frac{\partial}{\partial u_k}\mathcal{L}(u, \lambda, \gamma_1, ..., \gamma_{k-1}) = 2\hat{\Sigma} u_k - 2\lambda u_k \Rightarrow \hat{\Sigma} u_k = \lambda u_k.$$

Thus, it is clear that λu_k is the k-th eigenpair of $\hat{\Sigma}$. From this result, we see that PCA is analogous to the eigendecomposition of the covariance matrix. Then given a data matrix, $A \in \mathbb{R}^{m \times n}$ we can use the sample covariance matrix to estimate the covariance matrix in order to perform sample PCA. It is common to center the columns of the data matrix when performing PCA, in which case the sample covariance matrix is equal to,

$$S = \frac{\tilde{A}^\top \tilde{A}}{(n-1)} = V \Lambda V^\top,$$

where $\tilde{A}$ is the centered data matrix, $V = [v_1, ..., v_n]$ is a matrix whose columns are the eigenvectors of the covariance matrix, and Λ is a diagonal matrix of eigenvalues. Then it can be shown that v_k satisfies,

$$v_k = \underset{\substack{u \in \mathbb{R}^m, ||u||=1, \\ u^\top u_j = 0, j=1...k-1}}{\text{argmax}} \quad u^\top S u,$$

where v_k is the k-th sample PC direction and $u^\top S u$ is the sample variance of $u^\top x_i$ for $i = 1, ..., n$. These results are important in linking PCA to Singular Value Decomposition (SVD), a process which is detailed in Appendix A.2. Decomposing $\tilde{A}$ using SVD, we get $\tilde{A} = U \Sigma V^\top$ where,

- U is a $m \times m$ matrix of left singular values. This matrix contains the orthogonal unit vectors in the direction maximal variance.
- Σ is a $m \times n$ diagonal matrix of singular values. Each singular value is the amount of variance explained by each direction vector in U.
- V is a $n \times n$ matrix of right singular values. This matrix is the correlation between each variable and the orthogonal unit vectors. This is sometimes called the *rotations* matrix because it "rotates" the data onto the axes of the new basis.

Note that this Σ is different from the covariance matrix $\hat{\Sigma}$. Truncating the matrices U, Σ, and V produces a low-rank approximation of A. Inputting the SVD into the sample covariance matrix equation reveals the relation between the two decompositions:

$$\begin{aligned} S &= \frac{(U\Sigma V^\top)^\top (U\Sigma V^\top)}{(n-1)} \\ &= \frac{V\Sigma U^\top U\Sigma V^\top}{(n-1)} \\ &= V\frac{\Sigma^2}{(n-1)}V^\top. \end{aligned}$$

Thus, we have that $(\sigma_k^2/(n-1)) = \lambda_k$, where σ_k is the k-th singular value and λ_k is the k-th eigenvalue. Most computer programs, like R, will use SVD to perform PCA because of its interpretability and numerical stability. While the eigendecomposition of S and SVD of $\tilde{A}$ will produce the same results when using exact arithmetic, on a computer with rounding error, extremely small singular values will correspond to even smaller eigenvalues, so it is important to consider potential round-off error. The Lauchli Matrix shown in Figure 2.4 is a well known example of this phenomenon.

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ \varepsilon & 0 & \dots & 0 \\ 0 & \varepsilon & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & \varepsilon \end{pmatrix}$$

Figure 2.4: A generalized Lauchli matrix where ε is an arbitrarily small value.

Let L be a 4×3 Lauchli matrix and $\varepsilon = 1 \times 10^{-8}$. The SVD of L using R returns singular values $\sigma = (\sqrt{3}, 1 \times 10^{-8}, 1 \times 10^{-8})$; however, computing the eigenvalues of $L^\top L$ returns $\lambda = (3, 0, 0)$.

2.4 Robust Principal Component Analysis

Another dimension reduction method is robust principal component analysis (RPCA), a modification of traditional PCA that is more robust to outliers and corrupted data points. These corrupt data points and gross errors are becoming more common in modern data, hence the motivation behind RPCA [5]. RPCA provides a low rank approximation to the original data such that

$A_{m\times n} = L_{m\times n} + S_{m\times n}$, where L is a low-rank matrix and S is a sparse matrix. Once we have separated our data matrix in this fashion, we simply use L to approximate the original matrix. Calculating these matrices can be done by solving the optimization problem,

$$\min_{L,S} ||L||_* + \lambda||S||_1,$$

subject to $A = L + S$. Here, $||L||_* := \sum_i \sigma_i$ where σ_i are the singular values of L; this norm $||\cdot||_*$ is also known as the nuclear norm. $||S||_1 := \sum_{i,j} |S_{i,j}|$ and is also known as the ℓ_1-norm or element-wise norm of S and λ is a tuning parameter which is elaborated upon in Section 2.3.1. Since the rank of a matrix is equal to the number of singular values, minimizing the nuclear norm ensures a low-rank matrix. Similarly, minimizing the element-wise norm ensures a sparse matrix.

This problem can be solved by convex optimization using an augmented Lagrangian method (ALM) called prinipal component pursuit. This method is guaranteed to work even with a high-rank L or large errors in S and is at a cost not much higher than traditional PCA [5]. The Lagrangian function is given by,

$$\mathcal{L}(L, S, Y) = ||L||_* + \lambda||S||_1 + \langle Y, A - L - S\rangle + \left(\frac{\mu}{2}\right)||A - L - S||_F^2,$$

where Y is the matrix Lagrange multiplier, $\langle *, *\rangle$ denotes the inner product, $\mu = (\max(m, n)/(4||A||_1))$, and $||*||_F$ is the Frobenius norm. Let $\mathcal{S}_\tau(x) = \max(|x| - \tau, 0)$ denote the shrinkage operator; it can be shown that,

$$\underset{L}{\text{argmin}}\mathcal{L}(L, S, Y) = \mathcal{S}_{\lambda\mu}(A - L + \mu^{-1}Y).$$

Similarly, let $\mathcal{D}_\tau(x) = U\Sigma_\tau V^\top$ denote the singular value threshold operator; which retains the singular values greater than the threshold τ. It can be shown that,

$$\underset{L}{\text{argmin}}\mathcal{L}(L, S, Y) = \mathcal{D}_\mu(A - S - \mu^{-1}Y).$$

This function can be solved by the *alternating directions* method where we first minimize $\mathcal{L}(L, S, Y)$ with respect to L, as seen above, then minimize $\mathcal{L}(L, S, Y)$ with respect to S while fixing L, and then updating Y based on the minimizers above until a certain threshold is met [5].

2.4.1 RPCA Tuning Parameter

The tuning parameter, λ, controls the number of non-zero entries in the matrix S. It can be thought of as a penalty term against corrupted data points [9]. We can improve the performance of RPCA by adjusting λ. For example, if S is

known to be very sparse, increasing λ can recover a higher-rank matrix L [5]. However, as a rule of thumb, λ is typically set to $1/\sqrt{\max(n, m)}$; this is the case in programming languages like R. For our purposes, we decided to use this value as it is recommended in the literature and felt our results were sufficient without adjusting the parameter.

2.5 Autoencoders

Autoencoders are another unsupervised learning method that leverages the architecture of neural networks. In particular, autoencoders offer the ability to utilize the network structure in a fashion that is conducive to a greater degree of freedom when reconstructing data, since nonlinear inter-layer mappings allow for nonlinear learning. Autoencoders force a bottleneck within the network that compresses the original data inputs, which in turn is useful when learning the structure of the original data. If the data were indeed independent, compressing the data through said bottleneck and reconstructing it in a meaningful way would be quite difficult. Without this data compression, the model could easily memorize the input data which would lead to a model that overfits the data rather than providing a relevant reconstruction.

Note that PCA is a special type of autoencoder in which the network bottleneck is trivial, namely the structure of the network is linear in nature, meaning that there are no nonlinear activation functions present. As is the case with PCA, autoencoders must balance reconstruction error with overfitting.

A general autoencoder may be visualized through inter-layer and inter-node connections in Figure 2.5.

2.5.1 Artificial Neural Networks

In order to understand the underlying architecture of artificial neural networks, terminology similar to that in neuroscience is employed. In particular, artificial neural networks consist of neurons, similar to the human brain. This network is organized of layers of neurons, or processing units, which are interconnected. By definition, a deep neural network has at least two hidden layers, which are layers of neurons that are not the input layer nor the output layer. The input layer of the network consists of sensory neurons, which perform no processing of information but simply pass information in memory to a subsequent layer.

Between neurons are directed connections, meaning that information flows in one direction, and each connection has an associated weight within the network. A weight is simply a parameter, but training a neural network

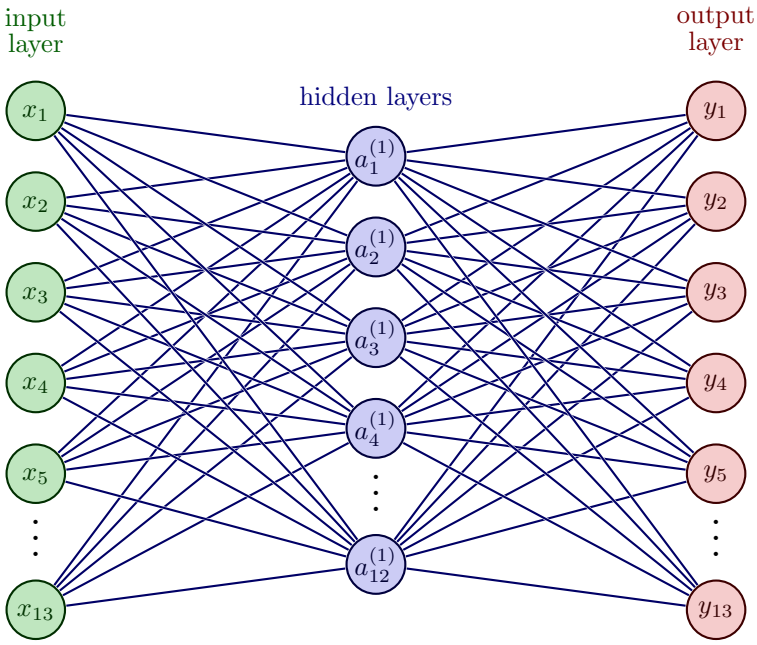


Figure 2.5: Fully connected autoencoder with a 13/12/13 node structure.

is predicated on choosing an optimal set of weights in order to achieve optimal results from the model. Neurons implement a two-stage process in order to map an input to an output:

1. Calculate a weighted sum of input parameters/values
2. This value is passed through another function that maps the weighted sum score to an output

The final output of a neuron is known as its activation value. The second function in this two-stage processing action is known as the activation function of the neuron.

A given neuron may receive n different inputs as a vector $[x_1, \ldots, x_n]$ along n different input connections in the network along with their associated weights as another n dimensional vector given by $[w_1, \ldots, w_n]$. The weighted sum is logically:

$$z = \sum_{i=1}^{n} w_i x_i.$$

A multitude of different activation functions have been tested, researched, and used in neural network research. Early research employed thresh-

old functions, followed by logistic and hyperbolic tangent functions. More modern functions that have allowed for better training of these networks include the rectifier (also known as hinge, or positive linear) activation function. Activation functions apply a nonlinear mapping from the weighted sum to the output, which is why they are extremely powerful tools.

Since the neural networks as a whole defines a map from inputs to outputs and neurons define the building blocks of the network, then the neurons build the mapping that the network itself defines. If all neurons within a network implement a linear mapping, then logically the network that is composed of these neurons must define a linear mapping. Many practical problems are nonlinear in nature, and leveraging a linear structure for a nonlinear relationship would yield a very inaccurate model. The attempt to force a nonlinear relationship to be approximated by a linear model would underfit the data and oversimplify the underlying structure. Theoretically, implementing any nonlinear activation function would allow the network to learn nonlinear mappings from inputs to outputs, but some functions have inherent properties that render them superior to other methods in terms of training and inference potential.

Processing for the network in done on an individual neuron level, and the behavior of the network as a whole is predicated on the interactions between each neuron. Once the weights on the connections between neurons are set, the network learns a decomposition of the original problem by having individual neurons learn their own respective solutions and how to combine solutions with other neurons. In practice, neurons may also be referred to as units, and these units are classified based on which activation function they carry. Some layers may have different activation functions relative to another layer, such as a layer of rectified linear units (ReLUs) connected to a layer of logistic units. Activation functions for neurons must be chosen for neurons before any processing is done.

The selection of activation functions is most often done by examining which models are the most popular and powerful at the given time. As of now, ReLUs are the most popular type of unit. Elements and properties of a network which are manually set beforehand are known as hyperparameters. It is critical to distinguish the hyperparameters from the parameters of the model, which are automatically set through the machine learning algorithm. To set these parameters for a neural network, weights are first initialized to random values. Training is subsequently performed in order to adjust the weights in order to achieve a more optimal performance by the model.

The firing of neurons can be easily visualized and analyzed via the introduction of a decision boundary, which is a graph of which possible combination of input parameters yield a firing of the neuron versus which parameters do not. All decision boundaries must pass through the origin, which is the origin of all weight vectors. Thus, changes in weights simply rotate the neuron's decision boundary rather than translating it. In order to subvert this possi-

ble limitation of the network, the weighted sum calculations include an extra parameter known as the bias term. This bias term allows for translations of the decision boundary. The output of a neuron is determined by passing the following z value into the chosen activation function for the neuron:

$$z = \left(\sum_{i=1}^{n} w_i x_i \right) + b.$$

Bias terms are typically calculated by having the neuron learn it similar to how it learns the respective weights for its inputs. Thus, all input vectors for the given neuron are always augmented with an additional input that is set to 1, namely input 0, or $x_0 = 1$.

2.5.2 General Architecture and Inter-Layer Computations

More explicitly in the case of the cryptocurrency data set obtained through the Binance API, autoencoders attempt to reconstruct the daily log returns across the 13 different assets. In particular, across the expanding window scheme, 826 data matrices are generated through the incremental concatenation of additional samples of returns data. We use autoencoders in our case to reconstruct the series of data matrices as $X^{(k)} = (x_1^{(k)}, x_2^{(k)}, \ldots, x_{64+k}^{(k)})^\top$, where $k = 1, \ldots, 826$ and each $x_i^{(k)}$ is a column vector with 13 entries corresponding to the daily log returns for each asset considered.

Thus, each $x_i^{(k)} \in \mathbb{R}^{13\times 1}$ is transformed into another vector in the intermediate hidden layer as $y_i \in \mathbb{R}^{12\times 1}$. This smaller dimension forces a compressed data representation and y_i corresponds to the bottleneck introduced in the network. This hidden layer is then decoded through the decoder mapping, yielding a data reconstruction given by $\hat{x}_i^{(k)}$.

In our case, we have a simple under-complete autoencoder, meaning that the number of nodes in the hidden layer is strictly less than the number of nodes in the input and output layers, respectively. The encoder mapping is given by the following:

$$y_i^{(k)} = \varphi_1(W_1 x_i^{(k)} + b_1).$$

This mapping is equipped with a linear or nonlinear activation function φ_1. In order to introduce nonlinear learning within the network, nonlinear functions are considered here, such as sigmoid, softmax, and tanh. Such functions are defined and graphed in Figure 2.6.

Algebraically, the weight matrix W_1 is a real-valued 12×13 matrix and b_1 is a 12×1 vector. After the encoder mapping has been applied to the data, a compressed data representation $Y^{(k)} = (y_1^{(k)}, y_2^{(k)}, \ldots, y_{12}^{(k)})^\top$ is obtained, corresponding to a data matrix with a reduced number of columns compared

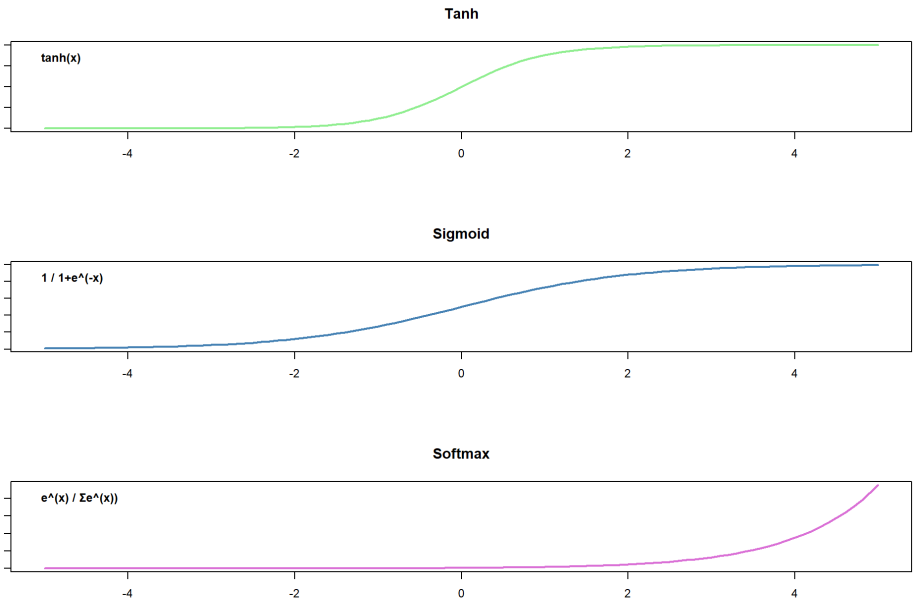


Figure 2.6: Graphs and equations of activation functions listed above

to $X^{(k)}$. Subsequently, the columns vectors $y_i^{(k)}$ outlined above are transformed via a decoder mapping given by

$$\hat{x}_i^{(k)} = \varphi_2(W_2 y_i^{(k)} + b_2),$$

where in this case, the weight matrix W_2 is a 13×12 real valued matrix and the bias vector b_2 is 13×1.

"Solving" the neural network is defined as determining the optimal weights and bias matrices such that the loss function is minimized. The loss function generally takes input values and compares their distance or proximity to the reconstructed output values. Thus, minimizing the loss function ensures that the output data representation is as close to the original input data as possible. In particular, the loss function in our network was defined as

$$L(W, b, X^{(k)}) = \frac{1}{2} \sum_{i=1}^{64+k} \left|\left| x_i^{(k)} - \hat{x}_i^{(k)} \right|\right|^2.$$

In order to perform dimension estimation using the autoencoder architecture, additional modifications must be made in practice. In particular, the modified loss function forces sparsity of small values in the intermediate hidden layer, where the data compression is performed, implicating that only important larger

values in $y_i^{(k)}$ are retained. Leveraging the methodology outlined by Bahadur and Paffenroth [11], the hidden layer vectors are normalized as follows

$$\mathfrak{y}_i^{(k)} = \frac{y_i^{(k)}}{||y_i^{(k)}||_2},$$

which ensures that the node values within the bottleneck hidden layer are consistent and able to be compared. Forcing the sparsity within the hidden layer is achieved through the inclusion of the so-called "regularizer value," or more specifically an L_1 regularizer λ. This is leveraged in the loss function by the addition of a term that penalizes insignificant terms, yielding sparsity in the layer. This new loss function is given by

$$\mathfrak{L}(W, b, X^{(k)}) = L(W, b, X^{(k)}) + \lambda \sum_{i=1}^{64+k} \left|\left|\mathfrak{y}_i^{(k)}\right|\right|_1.$$

In this case, W is the set of both weight matrices for the encoder and decoder mappings, b is the set of both bias vectors for the mappings, and $||*||_1$ is the L_1 norm defined in Section 2.3. The minimization of this $\mathfrak{L}$ function over the set of possible weight and bias matrices subject to the "optimal" choice of λ provides a set of reconstructed column vectors, $\hat{x}_i^{(k)}$, relative to the set of input vectors. In the subsequent analysis, two regularizer values of 1×10^{-5} and 5×10^{-5} are considered.

2.5.3 Singular Value Proxies

In order to perform dimension estimation, singular value analogues, which will be referred to here as singular value proxies as documented in Bahadur and Paffenroth [11], are found in the hidden layer bottleneck. In particular, the matrix as referenced above, $Y^{(k)} = (y_1^{(k)}, \ldots, y_{64+k}^{(k)})^\top$, is used to compute SVPs. As in singular value decomposition, the rows of this special $Y^{(k)}$ matrix are sorted in descending order from left to right. This new matrix is denoted by M. Then, averages along the columns of M are computed in order to determine the singular values, namely

$$svp_i = \frac{1}{64+k} \sum_{j=1}^{64+k} M_{j,i},\ i = 1, \ldots, 12.$$

Here, the classical notation is utilized for the j-th row and i-th column of M as $M_{j,i}$. Thus, the set $\{svp_d\}_{1 \leq d \leq 12}$ are known as the SVPs for the 12 features.

2.5.4 Hidden Layer Singular Value Decomposition

The existing framework outline above as documented in Stapleton's report provides the structure for performing a dimension estimation method known as hidden layer singular value decomposition [9]. In this way, taking the loss function $\mathfrak{J}$ above along with a 13/12/13 node structure, the intermediate matrix $Y^{(k)}$ is used by performing SVD on $(Y^{(k)})^\top$. In this way, the singular values are obtained and are able to be analyzed as in the PCA and RPCA methods.

Chapter 3

Methodology

Our goal is to find a method which simultaneously lowers the dimension of our data, preserves as much information as possible and is robust to market crashes and other anomalies. To assess the performance of each model, we will consider three factors: the estimated dimensions, loadings for PCA and RPCA and the root mean squared error between the original and reconstructed data sets.

3.1 Dimension Estimation

We have shown that it is possible to decompose a data matrix using PCA, RPCA, and Autoencoders into a new space of uncorrelated vectors. However, not all of these variables need to be retained. Let $T = AV$ be a transformation that maps a data matrix $A \in \mathbb{R}^{m \times n}$ to n variables that are uncorrelated over the data set and V is the matrix of eigenvectors of A. Keeping only the first k principal components results in the truncated transformation

$$T_k = AV_k,$$

where $V_k \in \mathbb{R}^{n \times k}$ is the truncated eigenvector matrix. This score matrix preserves the most variance in the original data set while minimizing the squared reconstruction error $||TV^\top - T_k V_k^\top||_2^2$. In the case of SVD, this score matrix can be written as

$$T = AV = U\Sigma V^\top V = U\Sigma,$$

and a truncated score matrix $T_k \in \mathbb{R}^{m\times k}$ can be obtained by considering only the first k singular values:

$$T_k = U_{m\times k}\Sigma_{k\times k}.$$

The truncation of a data matrix via SVD produces a matrix of rank k that minimizes the Frobenius norm between the truncated matrix and the original data matrix, which is a result of the Eckart-Young theorem. This is a useful result in reducing the dimensionality of a high-dimensional matrix, since it allows for the retention of the most possible variance from the data set.

For example, choosing to keep the first two principal components creates a plane in the direction that the high-dimensional data is spread out. Typically, these two components are not sufficient to explain most of the variance. However, choosing too many principal components may result in overfitting to the sample data set and defeats the purpose of dimension reduction.

This concept of selecting an appropriate number of principal components is surprisingly subjective. It is typically up to the discretion of the researchers conducting the analysis to determine what they feel is sufficient for their purposes. However, they may risk underestimation (a loss of information) or overestimation (retaining redundant information) by simply relying on intuition. There exist several different rules for selecting which principal components to keep, which try to minimise these risks. They are, in no particular order, Kaiser Rule, Ratio Rule, Variance Explained Criteria, and Horn's Parallel Analysis.

3.1.1 Kaiser Rule

The first rule for dimension estimation we will explore is the Kaiser Rule (sometimes referred to as the Kaiser-Guttman Rule or K1-Criterion). The rule states that all sample eigenvalues of the correlation matrix that are greater than the mean of the eigenvalues should be retained. We have shown earlier that each singular value of a data matrix is equivalent to the square root of the corresponding eigenvalue of the correlation matrix. That is, the Kaiser Rule says to choose each principal component for which

$$\sigma_d^2 \geq \frac{1}{j}\sum_{i=1}^{j}(\sigma_i^2),$$

where $j = 13$ in our case, since there are 13 assets. This rule is often criticised for overretention. On average, the Kaiser Rule overestimated the number of principal components by 66%. Guttman claimed that in PCA, since the total variance equals the number of variables, then in an infinite population, a

"weak" upper bound to the number of true principal components is equal to the components with eigenvalues greater than one [7].

When applying the Kaiser-Criterion to our data set, we will retain the singular values greater than their mean. When analysing the differences between the methods of Principal Component estimation, we expect this will provide us with an upper bound on the dimension estimation.

3.1.2 Variance Explained Criteria

The Variance Explained Criteria retains the first k principal components that account for a certain percentage of the variance in the data. Recall that the eigenvalues of a correlation matrix are equal to the amount of variance explained by the corresponding principal component. Thus this rule can apply this rule to SVD for some k such that

$$\sum_{i=1}^{k} \left(\frac{\sigma_i^2}{\sum_{j=1}^{13} \sigma_j^2} \right) \geq P,$$

where P is a percentage threshold chosen with an emphasis on parsimony. The typical choice for P is 90%; however, it can be as low as 50%. Clearly, P should never be 100% as that would not reduce the dimension of the data matrix. For our data, we will be using a value of $P = 85\%$, which we determined during preliminary PCA that it explained a sufficient amount of variance without overretaining.

3.1.3 Ratio Rule

The Eigenvalue Ratio Test for the number of factors was a method introduced recently in 2013 in the journal Econometrica by economists Seung Ahn and Alex Horenstein [1]. The method was devised specifically for financial data, thus we expect it should preform well for our purposes. The rule is calculated as follows:

$$ER(k) = \frac{\lambda_k}{\lambda_{k+1}},$$

where ER is the eigenvalue ratio and k is index of the eigenvalue, with k = 1 being the largest eigenvalue. The number of factors to retain, are every $k \leq \tilde{k}$, where

$$\tilde{k} = \arg\max ER(k).$$

3.1.4 Horn's Parallel Analysis

Horn's Parallel Analysis (PA) is typically regarded as the gold-standard for dimension estimation for PCA and factor analysis. PA compares the eigenvalues from the correlation matrix with eigenvalues generated from Monte-Carlo simulated matrices with identical dimensions to the original data matrix.

According to Horn, it is commonly assumed that non-correlated data will be perfectly non-collinear and expect eigenvalues equal to one in PCA. However, due to sampling error and least square bias, there exists multicollinearity even in non-correlated data. Thus, in a finite sample of m variables and n samples, the eigenvalues from PCA will be greater or less than one. Therefore, when making a decision of component retention, Horn argued that researchers should adjust the eigenvalues of the correlation matrix by subtracting the mean sample-error of a large number of randomly generated $n \times m$ data sets and retaining only components with adjusted eigenvalues greater than one. Notice that this makes PA a "sample-based adaptation of the population-based Kaiser rule." [7]

While PA is lauded as the best method for dimension estimation, it has historically been avoided by researchers. This is likely due to the computational complexity of generating a large number of high-dimensional data matrices. However, due to technological advances, PA is now widely accessible. The programming language R includes a built-in function, *paran*, which we intend to use for our analysis. The function starts by randomly generating $30 * n$ data sets based on the original data set, where n is the number of variables in the sample data set and computes the eigenvalues for each of these sets. Then values greater than one are retained in the adjustment,

$$\lambda_p - (\lambda_p^r - 1),$$

where λ_p is the p-th eigenvalue and λ_p^r is the corresponding mean eigenvalue of the simulated random data sets.

3.2 Loadings Analysis

Recall from Section 2.2, PCA decomposes the data matrix into three matrices, $U, \Sigma, V^\top$, where U is the scores matrix, Σ is a diagonal matrix of the square-root of eigenvalues and $V^\top$ is the loadings matrix. The loadings matrix can be though of as "how much" each variable contributed when calculating a principal component. More formally, they are the correlations between the original variables and the calculated component. Squaring these loadings gives the amount of each variable contributes to the component, which we can then analyze.

In R, we can simply get the loading matrix from the SVD data frame which is outputted from the *prcomp* and *rpca* commands. When plotting the loadings at each iteration of the expanding window scheme, we ideally want to see a constant contribution from each asset over time.

3.3 RMSE of Reconstruction

We have shown that the dimension of our data matrix can be reduced via the methods we have outlined, however we want to ensure that these methods preserve a sufficient amount of information from the original data set. To measure the information lost during the dimension reduction, we will consider the root mean squared error (RMSE) between the original data matrix and its low-rank approximation obtained via PCA, RPCA, and autoencoders. RMSE is a metric used to measure the differences between the original values of a data set and their corresponding estimated values. RMSE is defined as the square root of the mean squared error, which is the average of the squared differences between the original and estimated value.

In our case, let A be the original data matrix and $\hat{A}$ be the reconstructed data matrix. The RMSE between these two matrices is defined as,

$$\sqrt{\frac{\sum_{i=1}^{m}\sum_{j=1}^{n}(a_{i,j}-\hat{a}_{i,j})^2}{N}},$$

where $a_{i,j}$ corresponds to the entry in the i-th row and j-th column of A, and $\hat{a}_{i,j}$ is defined similarly for $\hat{A}$. Notice that the numerator of this expression is equal to the Frobenius norm of the matrix defined by $A-\hat{A}$. Therefore, we can rewrite this expression as,

$$\frac{||A-\hat{A}||_F}{\sqrt{N}}.$$

For PCA, we will need to manually reconstruct the data set after performing the method. We used the *prcomp* function in R, which returns a data frame of relevant matrices. We will multiply the original data set by the rotations matrix from this data frame, which is the matrix of variable loadings. Doing this will project the data into the new lower-rank space generated from PCA, which is functionally identical to reconstructing the truncated matrices in SVD. Next, given how RPCA is calculated, we will use the Frobenius norm of the sparse matrix as the reconstruction norm. Finally, for the autoencoder, the difference between the output layer and original data matrix is considered. Ideally, we want a method that produces the smallest error while also sufficiently reducing the dimension of the original data set.

Chapter 4

Results

The following figures are the results of implementing our methodology onto the daily log returns of our 13 assets. To calculate log returns, we use the equation

$$R_t = \ln\left(\frac{P_t}{P_{t-1}}\right),$$

where P_t is the closing price of the asset on day t.

4.1 Principal Component Analysis

We begin by performing PCA on the entire data set in order to understand how it works for a single iteration of the expanding window scheme (EWS). To accomplish this, we use the R command *prcomp* and assigned it to the entire data object. From the exploratory data analysis (EDA), we determined that the mean for all the assets was approximately 0 and all the standard deviations were of the same order, so we do not need augment the *scale* or *center* parameters of the function. A more in-depth explanation of our code can be found in Appendix B. We begin by determining the number of principal components to retain, which starts with observing the scree plot of the data.

Pictured in Figure 4.1 is the scree plot obtained from performing PCA. A scree plot shows the amount of variance explained by each principal component (PC). Specifically, each eigenvalue was divided by the sum of all the eigenvalues and they were plotting in descending order. From the plot, we can see that about 60% of the data is explained by the first PC before dropping off sharply to about 15% by the second PC and decaying to below 5% by the fourth PC. When examining a scree plot, the "elbow" of the graph is where it begins

to level off and eigenvalues to the left of this point are considered significant. Thus there are around four significant components and we would expect this to be reflected in the other dimension estimation criteria.

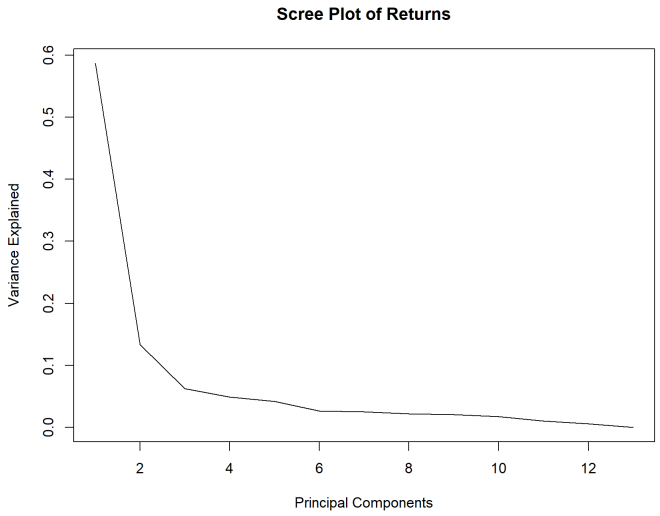


Figure 4.1: Scree Plot of PCA performed on the entire data set.

As we can see in Figure 4.2 the average number of PCs retained from the dimension estimation methods is four, with the ratio rule retaining the most with six PCs and Horn's parallel analysis retaining the fewest with one PC. These values support what we saw in the scree plot.

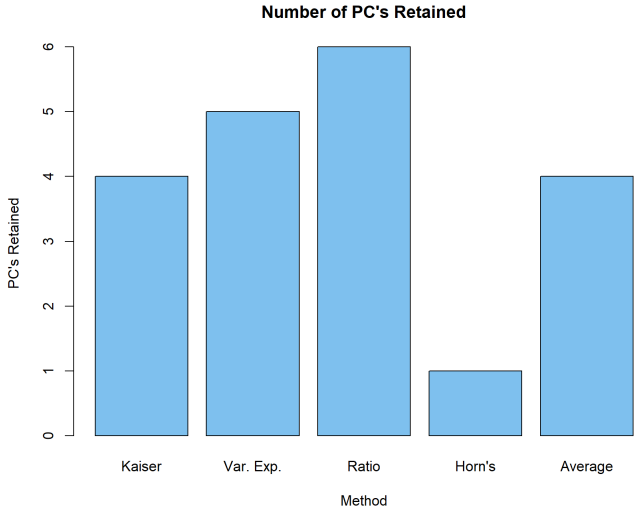


Figure 4.2: Principal components retained using different methods.

Next, we create a bar chart of the loadings as seen in Figure 4.3 to show the contributions of each asset to each PC. Notably, PC1 is approximately equally correlated to each asset excluding BUSD. However, BUSD accounts for almost the entirety of PC13. This is the last principal component, meaning it has the smallest contribution to the variance explained in the data.

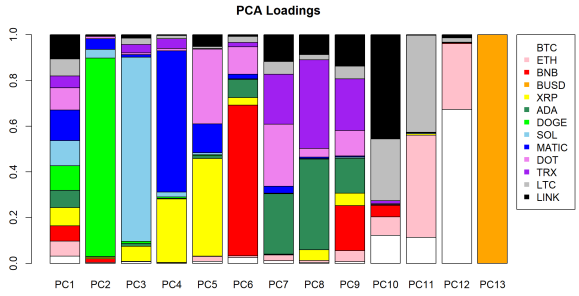


Figure 4.3: PCA loadings of the entire data set.

Now that we have a general understanding of the data set, we apply the EWS and create plots of the dimension estimation, loadings, and root mean squared error (RMSE) over each iteration. We plot the first four loadings because we identified that the first four principal components are the most sig-

nificant. We can see in Figure 4.4 that each dimension estimation criterion is relatively stable after iteration 200. Kaiser's rule and Ratio rule only have one jump in the estimated dimensions after iteration 200, which both happen close to iteration 590.

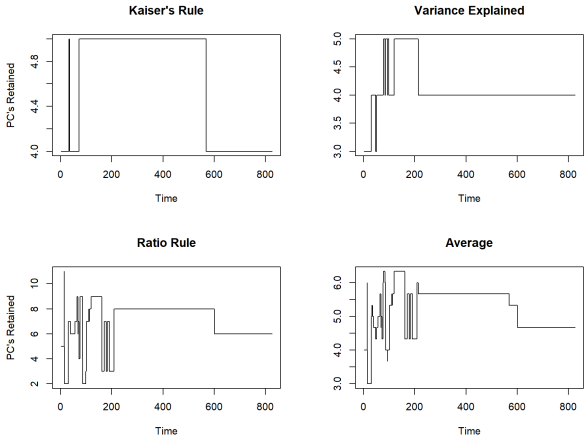


Figure 4.4: Principal components retained over time.

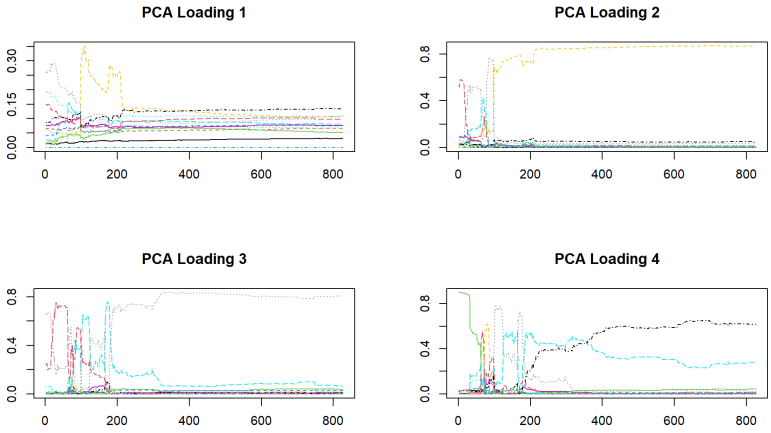


Figure 4.5: First four PCA loadings over time.

We see a similar trend in Figure 4.5. Each loading is violatile at the start of the expanding window scheme, however, there is stability in each of the loadings after iteration 200. Notably, there is a large spike in the yellow line, which is associated with Dogecoin, around iteration 100 in the first two

loadings. This spike is also present at the same iteration in figure 4.4. In figure 4.6, we see that the RMSE of the PCA reconstruction increases until plateauing at around iteration 175, then decays after iteration 200.

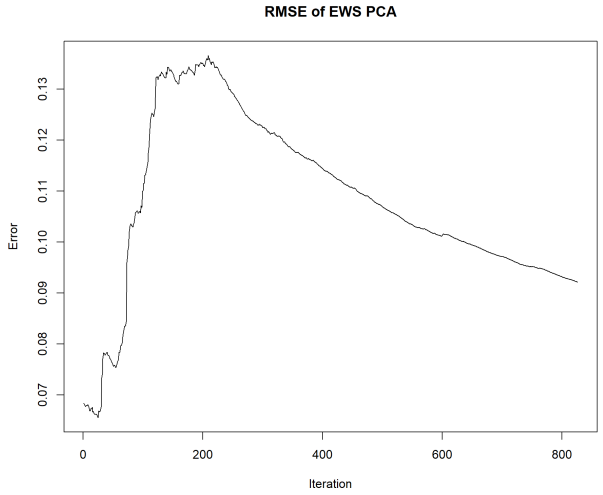


Figure 4.6: Root mean square error of PCA reconstrucution at each EWS iteration.

4.2 RPCA

Similarly to with PCA, we perform RPCA on the entire data set to understand each iteration of the EWS. We used R's *rpca* function and assigned it to the data set. The *rpca* function returns two matrices—L, the low rank matrix approximation, and S, the corresponding sparse matrix—and two data frames—$L.svd$ and $convergence$. Any reference to loadings are obtained from the $L.svd$ data frame's Vt matrix and the singular values come from the $L.svd$ data frame's D matrix. Plotting the variance explained from these singular values produces the scree plot in Figure 4.7. The scree plot shows that almost 100% of the variance is explained by the first PC, and the variance explained by PCs two through six are extremely close to zero. Components after six were all equal to zero, because the rank of L is 6.

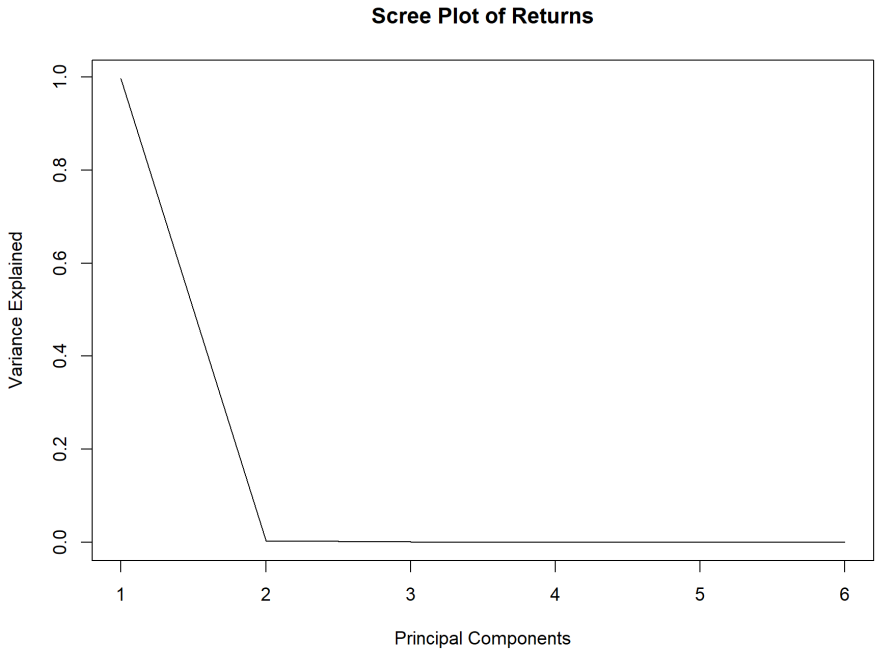


Figure 4.7: Screen Plot of RPCA performed on the entire data set.

Looking at the dimension estimation criteria, we see that both Kaiser's rule and Variance Explained rule retained only one PC, which is verified by the scree plot. The ratio rule retained the most components at 3, resulting in an average of 1.5 components. This is significantly fewer than the 4 components retained by PCA.

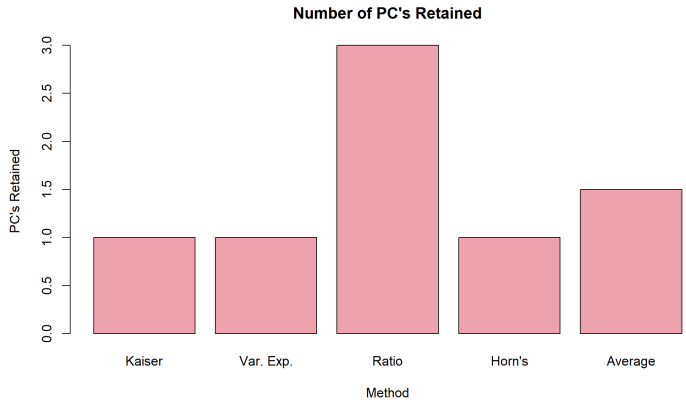


Figure 4.8: Principal components retained using different methods.

Further supporting the previous two results, the loadings in Figure 4.9 show that the correlations between the assets and the principal components are more evenly distributed than PCA. Aside from BUSD in PC6, there do not appear to be any principal components with high correlations to a single asset like there are in the PCA loadings.

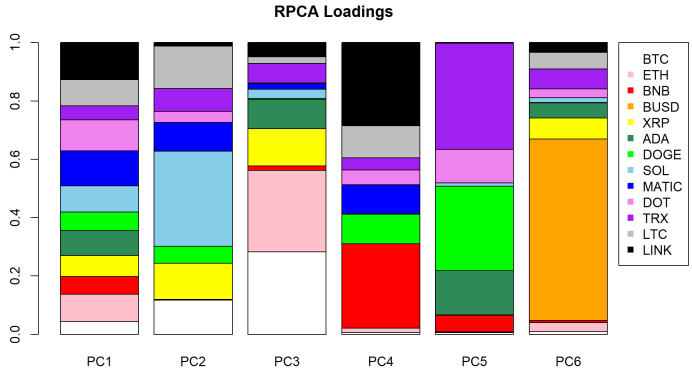


Figure 4.9: RPCA loadings of the entire data set.

After performing RPCA over the EWS, we plot the dimension estimation from the criteria in Figure 4.10. We can see that Kaiser's rule and Variance Explained rule are very consistent. Kaiser's rule only has one jump at iteration 308 from two PCs to one. Meanwhile, variance explained does not change at all and remains at one PC throughout the expanding window scheme.

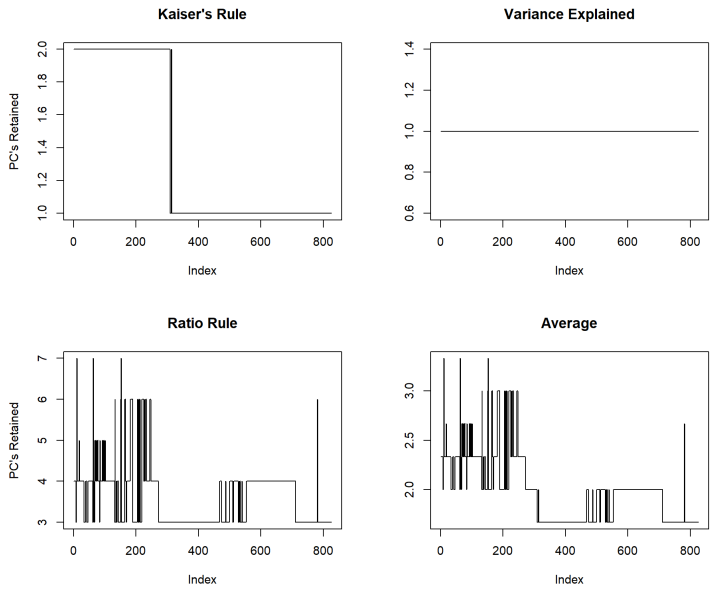


Figure 4.10: Principal components retained using different methods.

Shown in Figure 4.11 are the loadings over each iteration of the EWS. We can see that the loadings of PC1 are fairly consistent after iteration 200. The loadings of PC2 are also fairly consistent after iteration 200, but with significantly more noise.

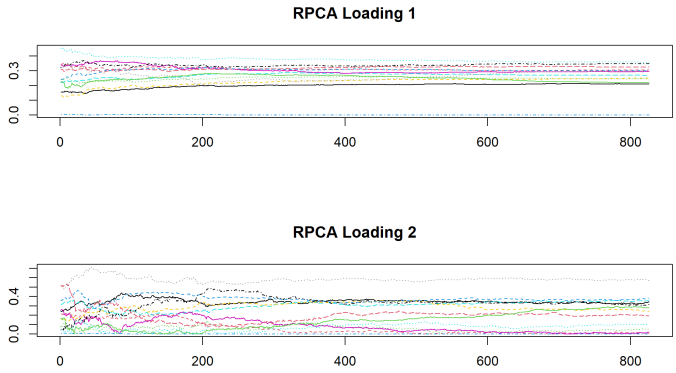


Figure 4.11: First four RPCA loadings over time.

The RMSE of RPCA is shown in Figure 4.12. It has a nearly identical curve and value as the RMSE of PCA, where it sharply increases until iteration 175, then levels off until 200 before decaying. The RMSE of RPCA peaks at 0.12 and approaches 0.08 at the final iteration, while the max RMSE for PCA is around 0.14 and approaches 0.09.

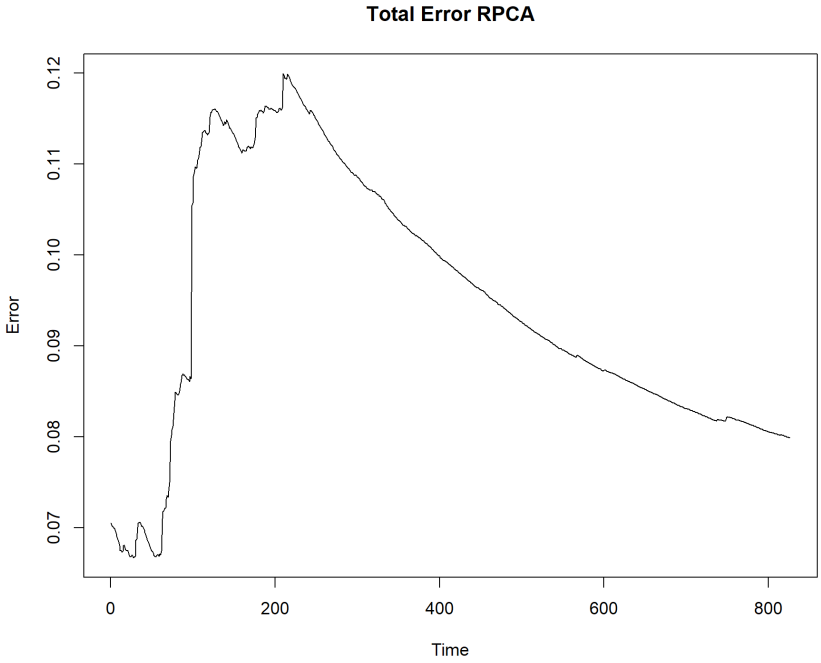


Figure 4.12: Root mean square error at each EWS iteration.

4.3 Autoencoders

For our autoencoder, we configured the hyperparameters by examining and comparing model performance in the scree plot and principal component retention graphs while also minimizing the average total loss over all trials. In particular, the initial autoencoder run attempted to tune the L_1 regularizer value with sigmoid and tanh activation functions for the hidden layer and output layer, respectively. Recall that the expanding window scheme iterates through $k = 1, \ldots, 826$. However, as the autoencoder is computationally expensive to run, a subset of these k values are identified using the anomalies found through the PCA and RPCA methods. Explicitly, we identified that $k = 98$, $k = 308$, and $k = 568$ corresponded with market crashes or anomalies[1]. In order to determine the robustness of the autoencoder surrounding these events, we looked at a neighborhood of ± 2 samples around these dates, and this corresponded to the set $\Omega = \{1, 96, 97, 98, 99, 100, 306, 307, 308, 309, 310, 566, 567, 568, 569, 570, 826\}$.

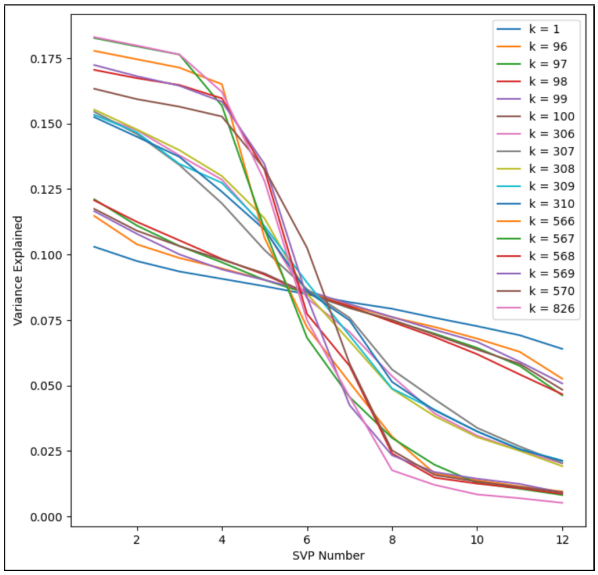
In the case of a 13/12/13 node structure equipped with a sigmoid/tanh set of activation functions for the respective layers along with a regularizer value of $\lambda = 5 * 10^{-5}$, the average total loss from the autoencoder over all trials was approximately 0.0368.

From the scree plot in Figure 4.13a and the graph of the average number of principal components retained of the selected iterations from the EWS in Figure 4.13c, it is clear that the model is not tuned completely efficiently. In particular, there are fluctuations in the principal component retention in the neighborhood surrounding the market anomalies and poor performance at lower k values as seen in the absence of the hockey stick shape in the scree plot.

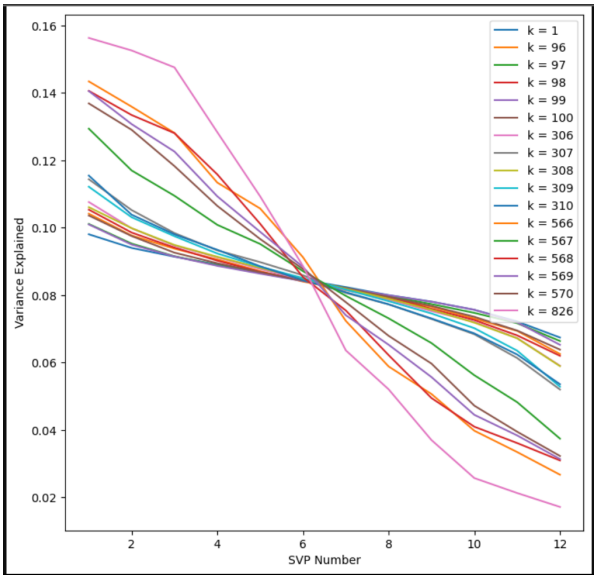
Holding constant the activation functions and node structure, the regularizer value was changed to $\lambda = 10^{-5}$ in order to assess whether model performance improved. Using this new value, the average total loss over all trials in this case was approximately 0.0119. Autoencoder performance as observed in Figure 4.13b and Figure 4.13d does not suggest any better or worse performance than the previous regularizer value. There is still substantial variation in principal component retention surrounding the k values of interest and a failure for the convergence of scree plots across k values.

Now considering a different pair of activation functions of softmax and tanh for the given layers, the model performance was significantly better compared to the sigmoid and tanh pairing. In particular, the average loss over all trials using a regularizer value of 10^{-5} was 0.005 while the average loss using a value of $5 * 10^{-5}$ was 0.0182. More importantly, however, there is a clear convergence of scree plots across the k values and also a robust principal

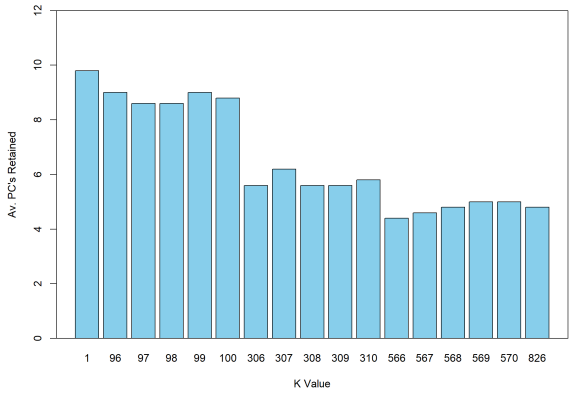
[1]Refer to Section 5.1 for notable dates that were identified and their associated market event.



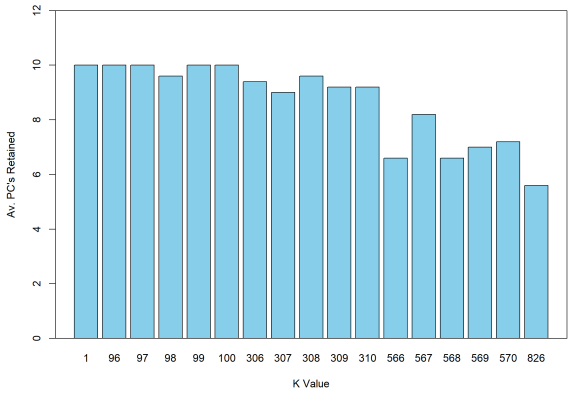
(a) Scree Plot over k values with $\lambda = 5 * 10^{-5}$.



(b) Scree Plot over k values with $\lambda = 1 * 10^{-5}$.

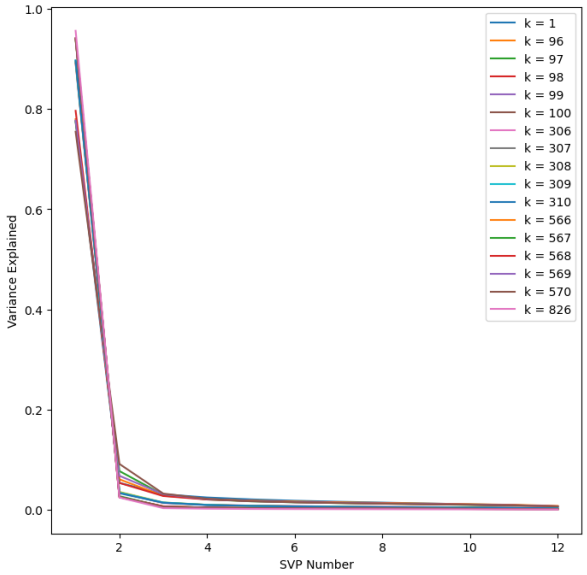


(c) Estimated dimensions over k values with $\lambda = 5 * 10^{-5}$.

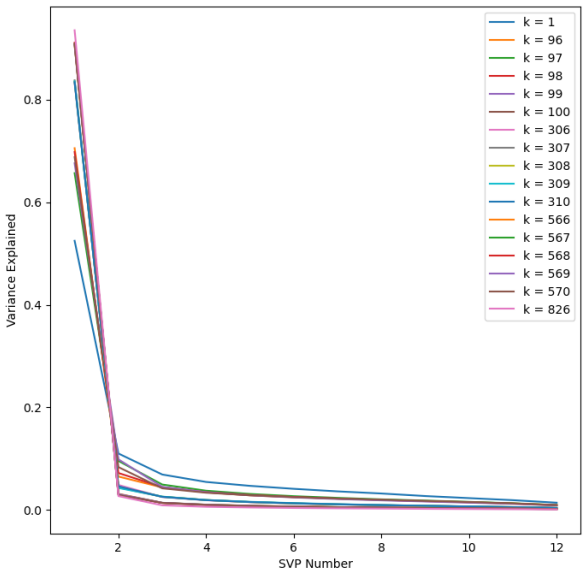


(d) Estimated dimensions over k values with $\lambda = 1 * 10^{-5}$.

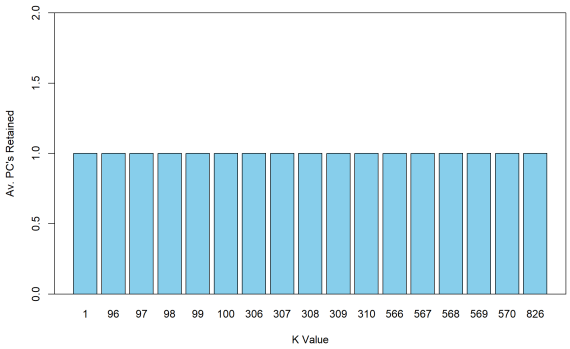
Figure 4.13: Autoencoder results with a 13/12/13 node structure equipped with sigmoid/tanh activation functions with different regularizer values.



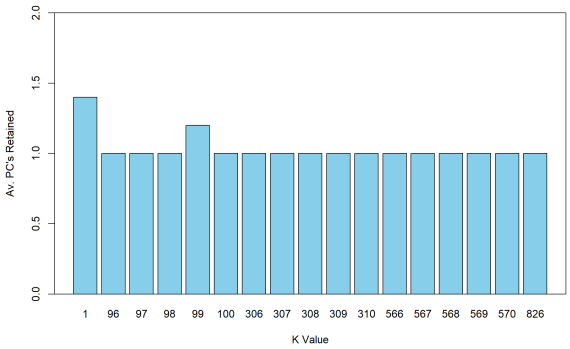
(a) Scree Plot over k values with $\lambda = 5 * 10^{-5}$.



(b) Scree Plot over k values with $\lambda = 1 * 10^{-5}$.



(c) Estimated dimensions over k values with $\lambda = 5 * 10^{-5}$.



(d) Estimated dimensions over k values with $\lambda = 1 * 10^{-5}$.

Figure 4.14: Autoencoder results with a 13/12/13 node structure equipped with softmax/tanh activation functions with different regularizer values.

component retention over time. Note that for each k value, the autoencoder was iterated five times in order to average any variability in learning and weight optimization. Thus, we see principal component retention of non-integer values such as at $k = 99$ as seen in Figure 4.14d, with an average of 1.2 principal components retained.

Despite having a higher average loss across trials, a regularizer value of $5 * 10^{-5}$ yielded more robust behavior over the EWS and had a tighter convergence in scree plots. Model robustness and consistency was maximized here, but overall reconstsruction error was lower with the alternative regularizer value. However, it is notable that performance with this new set of activation functions was clearly superior to the former choice. Thus, the autoencoder was optimized using a 13/12/13 node structure, softmax/tanh activation functions, and regularizer value of $\lambda = 10^{-5}$. Also note that these results employ the variance explained criterion using an 85% threshold variance.

Also note that the simplistic single layer autoencoder (SLAE) employed here may be generalized further to an n-layer deep autoencoder. This general architecture was initially employed through a 13/12/11/12/13 node structure in order to have a symmetric information structure for the encoder and decoder mapping through the bottleneck. However, its efficacy was highly questionable due to the incredible demand on a large number of price observations of the asset set. In particular, since our 13 assets only had 890 observations, the deep autoencoder architecture was foregone, since its performative improvements in relation to the SLAE were not realized due to this lack of data. With each additional layer added, the demand on more data for effective learning is immense. Despite this, however, our autoencoder with only one hidden layer performed relatively well in comparison to the other dimension reduction methods.

Chapter 5

Analysis

In this chapter, we analyze the results of our research and methods. We first identify dates corresponding to iterations of the expanding window scheme (EWS) where anomalies occurred and connect those anomalies to real-world events. We then compare the methods used to estimate the dimensionality of the data. Finally, we interpret our results in the context of the cryptocurrency market and investment portfolio diversification.

5.1 Notable Dates Identified

When performing PCA and RPCA, we noticed several dates when there are jumps in dimension estimations or loadings. Since we want a consistent number of components retained and smooth loadings, we used these dates to check if the autoencoder was more robust to these anomalies. Here, we determine if these spikes correspond with notable events involving cryptocurrencies. If a certain type of event is associated with these dates and causes a spike for a certain method but not another, we can conclude that one method is more robust to those types of events. We observe three significant spikes at iterations 97, 308, and 567. These correspond to the following dates: January 27, 2021; August 25, 2021; and May 12, 2022.

5.1.1 January 27, 2021

There were two significant events that happened around this date. On January 27, 2021, Elon Musk—the founder of Tesla and known proponent of cryptocurrencies—tweeted "#bitcoin" with no explanation [20]. This was after a particu-

larly poor month for cryptocurrencies that was exacerbated by a massive sell-off that wiped out nearly $100B from the cryptocurrency market earlier that week. Musk's tweet caused a resurgence in Bitcoin's market value, causing a single day increase of 16% [20]. This jump is clearly visible in figure 2.1, where the mean of BTC increases considerably.

Also on this day, users of a Reddit board joked about making Dogecoin the next "GameStop Stock." [14] Elon Musk tweeted a picture of a magazine cover of "Dogue," a parody of the magazine Vogue, and many people saw this as him supporting the rally. This ultimately resulted in an 800% increase in Dogecoin's market value [14]. This can be seen in Figure 4.5, where Dogecoin is represented by the yellow line. This is likely responsible for the change in dimension of PCA near this iteration, meanwhile RPCA was robust to this outlier and retained a consistent number of components.

5.1.2 August 25, 2021

There is no single event related to cryptocurrencies around this date. However, there was a strong bull market in late July and August that led to some of the highest market values in several months. Most asset prices have a local maximum at the iteration corresponding to this date. Most notably, Solana's (SOL) value grew nearly 200% in August [18]. This increase is visible in the third loading of PCA where a large jump in SOL (light grey line) can be seen. A similar, but smaller, jump in SOL can also be seen around $k = 200$ in the second loading of RPCA.

However, there were new regulations around this time in China that forced some of the largest cryptocurrency miners to shut down [18]. The United States also passed legislation that taxed cryptocurrencies [18]. After this legislation passed, there was a significant decrease in cryptocurrency value, which is visible in the price data in Appendix C. It is unclear which of these events may have caused the jump in the dimension estimation of RPCA; however, since all the assets were performing well at this time, there may have been an increase in correlation between them that caused this spike.

5.1.3 May 12, 2022

On this day, there was a massive $200 billion sell-off from the entire cryptocurrency market [2]. This sell-off caused the market value of multiple coins to plummet. For example, Bitcoin hit a 16-month low. It is believed that this sell-off was tied to broader economic insecurity, especially the inflation of the price of consumer goods and decline of the stock market [2]. RPCA is significantly more robust to this market crash than PCA. There is little change in the

dimension estimation or loadings at this date for RPCA; however, this event is apparent in PCA dimension estimation.

5.2 Method Comparison

From section 5.1, we can see that PCA was the least robust method during market anomalies, particularly large sell-offs or spikes in value. Throughout the two largest market crashes, RPCA and the autoencoder both returned consistent dimension estimations and RPCA's loadings remained smooth. RPCA's only jump in dimension estimation was at $k = 308$ using Kaiser's rule when the assets may have been highly correlated, however this jump is not present in the other criteria or methods. Thus, considering only the variance explained criterion, RPCA and the autoencoder performed the same in terms of robustness to market crashes.

Overall, PCA performed the worst across all the metrics we considered. PCA had the highest reconstruction error and was the least robust to anomalies, which is particularly visible in its loadings where there is a large spike associated with Dogecoin's anomalous behavior. Meanwhile, the autoencoder and RPCA performed similarly in terms of dimension estimation however, the autoencoder had a significantly lower reconstruction error. While the computing times were not formally calculated in this project, it is worth noting that the autoencoder took significantly longer to compute than RPCA. While RPCA only took minutes to calculate the entirety of the expanding window scheme, the autoencoder would theoretically take several hours to days to do the same; hence why we only considered a few select iterations.

Examining the different dimension estimation criteria used in the PCA and RPCA outlined above, there is a clear difference in the variability in the number of principle components retained on an inter-criteria scale. For PCA, Kaiser's rule provided minimal variations in its principal component retention, but was subject to variability surrounding exogenous cryptocurrency market shocks. However, since PCA is highly corrupted from outliers, the merits of the dimension estimation criteria should be analyzed relative to the RPCA results. Here, we see that Kaiser's rule was not as robust to market anomalies as seen in the drop in principal component retention around iteration $k = 308$ in the EWS. The variance explained criteria was constant across the window, providing consistent and robust estimations. Similar to PCA, however, the ratio rule was highly volatile over time.

One hypothesis regarding the variability exhibited in the ratio rule across PCA and RPCA is that its definition is such that it is highly subject to variations when there exist large discrepancies between singular value magnitudes. If singular value magnitudes drop significantly and are substantially

smaller across different iterations of the EWS, then the ratio between singular values could be maximized at significantly different indices for σ_i^2 for $1 \leq i \leq 12$. Given this pitfall with the ratio rule and the propensity for Kaiser's rule to be influenced by exogenous market shocks, we recommend the variance explained criterion with a threshold determined based on exploratory data analysis. In particular, this method exhibited the highest levels of stability. This is likely because attempting to explain some percentage threshold variance in the context of cryptocurrencies and other securities in general is the most well-defined metric of those considered.

5.3 Cryptocurrency Market

The cryptocurrency market is very volatile, but according to our findings, most of the variance can be explained with one or two components. Through the use of our most robust methods—RPCA and autoencoders—we find that on average, using the variance explained criteria, we retain 85% with just one principal component (PC). In addition, the inclusion of the second PC explains over 99% of the variance, indicating that there is a high degree of correlation among the 13 cryptocurrencies. Furthmore, looking at the first loading reveals that most of the assets vary in the same direction; but it is worth noting that Solana (SOL) appears to explain a large amount of the second principal component in RPCA and the third in PCA. Similarly, the last loading vector in each method is comprised almost entirely of BUSD. This means that both these two coins vary differently from the majority of the assets.

However, since the first principal component in all the methods considered explains almost all the variance in the data, we can conclude that all of the assets are highly correlated, and these cryptocurrencies do not lend themselves to being part of a diverse portfolio. The 13 cryptocurrencies included in our data represent a majority of the value in the cryptocurrency market; thus, the risk provided by the volatility of the cryptocurrency market cannot be avoided by making diverse investments.

Chapter 6

Conclusions

We compared the robustness of principal component analysis (PCA), robust principal component analysis (RPCA), and autoencoders in estimating the intrinsic dimension of the daily log returns of 13 cryptocurrency assets. Understanding the dimensionality of the data can provide insight into correlation between assets, which is important when balancing concentration risk and diversification. Having a method that is robust to market anomalies is especially useful when considering a highly volatile market like cryptocurrencies.

Regarding stability between methods, we found that PCA performed the worst, as it was the least robust to market crashes and its dimension estimation varied the most. The most robust method in this regard appears to be the autoencoder, which consistently retained one component when analyzing the market anomalies. The autoencoder also yielded the lowest reconstruction error. RPCA performed almost identically to the autoencoder in terms of dimension estimation, but vastly outperformed it in computational time and interpretability. Therefore, based on these results we recommend RPCA as the best of the three methods for dimension estimation.

When analyzing dimension estimation criteria, we found the ratio rule criterion exhibited high levels of variation in estimated dimension—especially at lower iterations—while the variance explained criterion was the most stable. This is likely because attempting to explain some percentage threshold variance in the context of cryptocurrencies and other securities in general is the most well-defined metric of those considered. Therefore, we recommend that the variance explained criterion should be weighted the most in the analysis of method performance.

The variance in the our crytocurrency data set can be explained by at most five dimensions. However, results from RPCA and autoencoders sug-

gest that it may have lower dimensionality; both of these methods retained an average of less than two dimensions. This is in line with previous literature that analyzed the connectedness of cryptocurrency and found that during times of exogenous market shocks, the connectedness increased [27]. Further, it was found that cryptocurrencies with high market capitalizations, like the ones we analyzed, tend to propagate volatility in the market and smaller coins followed [27]. This connection is likely why the intrinsic dimension of our data set was very low.

In conclusion, the low dimensionality of the cryptocurrency data set suggest that the assets have very similar variation, meaning that when building a cryptocurrency portfolio, diversification is extremely difficult. Thus, when selecting from the 13 assets explored here, the selection is essentially arbitrary.

Bibliography

[1] Seung C. Ahn and Alex R. Hornstein. Eigenvalue ratio test for the number of factors. *Econometrica*, 81(3), 2013. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2942796.

[2] Ryan Browne. More than $200 billion erased from entire crypto market in a day as sell-off intensifies. *CNBC*, 2022. https://www.cnbc.com/2022/05/12/bitcoin-btc-price-falls-below-27000-as-crypto-sell-off-intensifies.html.

[3] Jason Brownlee. Evaluate the performance of deep learning models in keras, 2022. https://machinelearningmastery.com/evaluate-performance-deep-learning-models-keras/.

[4] Ovidiu Calin. *Deep Learning Architectures A Mathematical Approach*. Cham : Springer International Publishing : Imprint: Springer, 2020.

[5] Emmanuel J. Candes, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *Journal of the ACM*, 58(3), 2009. https://arxiv.org/abs/0912.3599/.

[6] Francois Chollet. *Building Autoencoders in Keras*. Keras, 2016. https://blog.keras.io/building-autoencoders-in-keras.html.

[7] Alexis Dinno. Exploring the sensitivity of horn's parallel analysis to the distributional form of random data. *National Library of Medicine*, 2009. https://pubmed.ncbi.nlm.nih.gov/20234802/.

[8] Neil Gandal, JT Hamrick, Tyler Moore, and Tali Oberman. Price manipulation in the bitcoin ecosystem. *Journal of Monetary Economics*, 95, 2018. https://www.sciencedirect.com/science/article/pii/S0304393217301666.

[9] Nicholas Gawron, Gracie Johnson, Austin Sibu, and Aaron Stapleton. *Decorrelation Detection in a Financial Time Series Data Set*. Worcester Polytechnic Institute, 2021. https://labs.wpi.edu/cims/project-portfolio/reu-2021-decorrelation-detection-in-a-financial-time-series-data-set/.

[10] Xinyu Huang, Weihao Han, David Newton, Emmanouil Platanakis, Dimitrios Stafylas, and Charles Sutcliffe. The diversification benefits of cryptocurrency asset categories and estimation risk: pre and post covid-19. *The European Journal of Finance*, 2022. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3894874.

[11] IEEE International Conference on Big Data. *Anomaly Detection in Exchange Traded Funds*. IEEE, 2020. https://ieeexplore.ieee.org/document/9378452.

[12] Ian T. Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions A*, 374(2065), 2016. https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0202.

[13] John Kelleher. *Deep learning*. Cambridge : MIT Press, 2019.

[14] Arjun Kharpal. Reddit frenzy pumps up dogecoin, a cryptocurrency started as a joke. *CNBC*, 2021. https://www.cnbc.com/2021/01/29/dogecoin-cryptocurrency-rises-over-400percent-after-reddit-group-talks-it-up.html.

[15] Donald E. Knuth, Tracy Larrabee, and Paul M. Roberts. *Mathematical Writing*. Mathematical Association of America, 1987. https://jmlr.csail.mit.edu/reviewing-papers/knuth_mathematical_writing.pdf.

[16] Ladislav Kristoufek. What are the main drivers of the bitcoin price? evidence from wavelet coherence analysis. *PLoS ONE*, 10(4), 2015. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0123923.

[17] Nir Kshetri and Jeffrey Voas. Blockchain in developing countries. *IEEE IT Professional*, 20(2), 2018. https://ieeexplore.ieee.org/document/8338009.

[18] Lesia M. Crypto news update — August 2021. *Medium*, 2021. https://medium.com/coinmonks/august-2021-crypto-news-update-2cc71bd24b27.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. https://bitcoin.org/bitcoin.pdf.

[20] Yakob Peterseil and Joanna Ossinger. Speculative frenzy spills into crypto as bitcoin tests highs. *Bloomberg*, 2021. https://www.bloomberg.com/news/articles/2021-01-28/bitcoin-s-wild-ride-accelerates-with-push-back-above-33-000.

[21] Sakesh Pusuluri. Autoencoders, 2020. https://medium.com/@sakeshpusuluri/autoencoders-52c81a6f1ae1.

[22] David Ruppert and David S. Matteson. *Statistics and Data Analysis for Financial Engineering with R examples*. New York, NY : Springer New York : Imprint: Springer, 2015.

[23] Huma Saeed, Hassaan Malik, Umair Bashir, Aiesha Ahmad, Shafia Riaz, Maheen Ilyas, Wajahat Anwaar Bukhari, and Muhammad Imran Ali Khan. Blockchain technology in healthcare: A systematic review. *PLoS ONE*, 17(4), 2022. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0266462.

[24] Gilbert Strang. *Linear Algebra and Learning From Data*. Wellesley - Cambridge Press, 2019.

[25] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*. SIAM, 1997.

[26] Imran us Salam. Autoencoders — guide and code in tensorflow 2.0, 2019. https://medium.com/red-buffer/autoencoders-guide-and-code-in-tensorflow-2-0-a4101571ce56.

[27] Shuyue Yi, Zishuang Xu, and Gang-Jin Wang. Volatility connectedness in the cryptocurrency market: Is bitcoin a dominant cryptocurrency? *International Review of Financial Analysis*, 60, 2018. https://www.sciencedirect.com/science/article/pii/S1057521918304095#s0080.

Appendix A

Preliminary Background

A.1 Eigendecomposition

Eigendecomposition is a matrix factorization, whereby a matrix is represented by its eigenvectors and eigenvalues. A matrix can only be factored in this way if it is diagonalizable. A square matrix is diagonalizable if and only if there exists an invertible matrix P and diagonal matrix D such that $P^{-1}AP = D$.

Definition (Eigenvectors and Eigenvalues): Assume $A \in \mathbb{R}^{n\times n}$, then there exist $m = \text{rank}(A)$ distinct, nonzero vectors x_i known as eigenvectors, that satisfy $Ax_i = \lambda_i x_i$, where λ is an eigenvalue corresponding to the i-th eigenvector.

To find the eigenvalues of A, we start by rewriting the equation $Ax = \lambda x$ as $Ax - \lambda x = 0$, which can be factored as $(A - \lambda I)x = 0$. Thus, the eigenvectors make up the nullspace of $A - \lambda I$. If $A - \lambda I$ has a non-zero solution, $A - \lambda I$ is not invertible and has determinant 0. This fact results in the characteristic polynomial of A, $p(\lambda) = \det(A - \lambda I) = 0$. This polynomial will have m distinct solutions with $1 \leq m \leq n$, which are the eigenvalues of A. For each eigenvalue, solving $(A - \lambda I)x = 0$ yields the corresponding eigenvector x.

Now that the definitions of eigenvectors and eigenvalues are established, the factorization of A can be derived from their properties:

$$Ax = \lambda x,$$

$$AX = X\Lambda,$$

$$AXX^{-1} = X\Lambda X^{-1},$$

where $X \in \mathbb{R}^{n\times n}$ whose i-th column corresponds with eigenvector x_i of A and Λ is a diagonal matrix whose diagonal elements are eigenvalues such that $\Lambda_{i,i} = \lambda_i$.

If each eigenvector corresponds to a distinct eigenvalue, then X is a square matrix and its columns are linearly independent, which means that X is invertible and A can be decomposed as

$$A = X\Lambda X^{-1}.$$

A.2 Singular Value Decomposition

Singular value decomposition (SVD) is a matrix factorization that generalizes eigendecomposition to any matrix. Let $A \in \mathbb{R}$ be an $m \times n$ matrix, then there exists three matrices U, V, and Σ such that $A = U\Sigma V^\top$ where U and V are both orthonormal matrices and Σ is a diagonal matrix of singular values. The singular values of A are unique and the number of singular values is equal to the rank(A) [24].

One choice to determine orthogonal vectors U and V for the SVD of A is to consider the eigenvectors of the matrices $AA^\top$ and $A^\top A$. Since $AA^\top$ and $A^\top A$ are square, symmetric and positive semi-definite their eigendecompositions exists. Let $\{u_i\}$ be the set of eigenvectors of $AA^\top$ and $\{v_i\}$ be the set of eigenvectors of $A^\top A$.

$AA^\top$ and $A^\top A$ will be symmetric—but not necessarily equal—matrices, thus their eigenvectors will form two distinct orthogonal sets, which can then be normalised. The **u** vectors are called the left singular vectors and the —V's are called the right singular vectors. The singular values, σ, are the square roots of the equivalent eigenvalues of $AA^\top$ and $A^\top A$ [24]. Then from our choices thus far we have that,

$$\begin{aligned} AA^\top u_i &= \sigma_i^2 u_i, \\ A^\top A v_i &= \sigma_i^2 v_i, \\ A v_i &= \sigma_i u_i. \end{aligned}$$

The vectors u_i with $1 \leq i \leq r$ form an orthonormal basis for the column space of A, while the remaining vectors u_i with $r < i \leq m$ form an orthonormal basis for the left nullspace of A. Similarly, the vectors v_i with $1 \leq i \leq r$ form an orthonormal basis for the row space of A, while the remaining vectors v_i with $r < i \leq n$ form an orthonormal basis for the right nullspace of A. Thus, including all vectors u and v in U and V, they become square, orthonormal matrices such that $AV = \Sigma U$. Since V is square and orthonormal, $V^{-1} = V^\top$. Thus, we have that $A = U\Sigma V^\top$.

A.2.1 SVD vs Eigendecomposition

Singular Value Decomposition does not suffer from the shortcomings of eigendecomposition. For example, in eigendecomposition, eigenvectors are not always orthogonal, there are not always enough eigenvectors to diagonalize, and A must be square. Meanwhile SVD is applicable to any matrix. Furthermore, the singular values of a matrix are more stable than its eigenvalues. Consider the 3×3 matrix below:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

The eigenvalues of this matrix are $\lambda_i = 0$ for $i \in \{1, 2, 3\}$. The singular values are $\sigma_1 = 1$, $\sigma_2 = 1$ and $\sigma_3 = 0$. Now, if an extremely small change is made to a single cell, the instability of eigenvalues can be seen:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1/100000 & 0 & 0 \end{pmatrix}.$$

The eigenvalues now become $\lambda_1 = 0.0215$, $\lambda_2 = -0.01 + 0.018i$, and $\lambda_3 = -0.0107 - 0.018i$, which are significantly different from the original eigenvalues. Meanwhile, the singular values have only changed by the same amount the matrix was changed; they are $\sigma_1 = 1$, $\sigma_2 = 1$, and $\sigma_3 = 1 \times 10^{-}6$.

A.3 Introduction to Machine Learning and Deep Learning

In Deep Learning, a common goal is to analyze and understand data and its underlying structures and themes. In particular, functions which are extracted from data sets in the Deep Learning contexts are neural networks. Put trivially, a Neural Network encodes functions as programs on a computer. When working with data in the hands-on context, two problems arise quite quickly:

1. Most datasets include noise, so having the model learn the dataset is not the best option for modeling and inference purposes.
2. In some cases, there is a lack of information that allows for a single best solution to be selected, but instead, a set of multiple solutions may fit the data.

In machine learning, we can aid the machine in selecting the best solution function for the dataset by making assumptions about the preferred characteristics

of the best function, known as the inductive bias. Different algorithms that are leveraged in practice encode different inductive biases. When choosing a machine learning algorithm with the incorrect bias, if the bias is too strong, then the function may underfit the data and ignore important or true underlying patterns which are present. Alternatively, if the bias is too weak, then the algorithm may find a solution that overfits the data nad firts the function to the noise that is present within the training subset, rendering any inference potential of this solution to be very weak and unpredictable. Neural networks have a relatively weak inductive bias, and are consequently prone to overfitting data. Thus, it is advantageous to implement these structures when working with large datasets since they are trained to rely heavily on the data in which they are supplied.

In machine learning, three critical pieces are required in order to successfully and logically create a model:

1. Data
2. A set of functions that must be considered as solutions by the algorithm in order to choose the best one
3. A measure of fitness in order to determine which function is the best fit for the data

When beginning to look at a machine learning model, it first must be established which features should be included within the data. If too few features are present, then an informative and important feature may be missing from the model, rendering it inaccurate. If, on the other hand, too many features are selected, then unimportant or inconsequential features may be present within the solution, thus hurting the algorithm as it becomes too prone to picking up spurious patterns within the data. Next, selecting which function representation for the data is appropriate is crucial when looking to create a machine learning model, for example, employing neural networks as a flexible and powerful function representation for big data. When changing how data is represented, the set of candidate function solutions, and the measure of fitness for the model, we obtain three different types of machine learning: supervised, unsupervised, and reinforcement learning.

Appendix B

Code

B.1 R Code

```
#Code for Nonlinear Dimension Estimation for Cryptocurrency Data
#By: Ben Rajotte


# Variables and Libraries ---------------------------------------------------
#Load the Libraries and Variables need for the code to function
library(lubridate)
library(MASS)
library(paran)
library(moments)
library(corrplot)
library(rpca)
library(readxl)
CD <- read_xlsx("APIData_Updated_SET_LOGReturns.xlsx") #Load data set
CR <- as.data.frame(CD)[-1] #Isolate returns

col_num = ncol(CR)
row_num = nrow(CR)

win_size = 64 #Set window size
expanding = TRUE #This value toggles between Expanding and Rolling Window

# Data Summary --------------------------------------------------------------
#Provides a general summary of the entire data set
```

```r
data_summary <- apply(CR, 2, summary)
data_summary <- rbind(data_summary, apply(CR,2,sd), apply(CR,2,skewness), apply(CR,2,kurtosis))
rownames(data_summary)[7:9] <- c("Std. Dev.", "Skew", "Kurt")

for (i in 1:col_num){
  asset_name = colnames(CR)[i]
  asset_ts <- ts(CR[asset_name], start = decimal_date(as.Date("2020-08-18")), frequency = 365.25)

  layout(matrix(c(1,1,2,3), nrow=2, byrow=TRUE))
  plot(asset_ts, xlab = "Date", ylab = asset_name, main = paste(asset_name,"Returns"))
  boxplot(CR[asset_name], horizontal = TRUE, main = asset_name, xlab = "Returns")
  acf(CR[asset_name], lag.max = 40)
}

# Preliminary PCA ----------------------------------------------------------

#PCA
pca_CR <- prcomp(CR)
var_exp_CR <- pca_CR$sdev^2 / sum(pca_CR$sdev^2)
plot((1:length(var_exp_CR)), var_exp_CR, "l",main = "Scree Plot of Returns",
xlab = "Principal Components", ylab = "Variance Explained")

#Dimension Estimation
dim_est_CR <- matrix(ncol = 5, nrow = 1)

dim_est_CR[1] <- length(which(pca_CR$sdev > mean(pca_CR$sdev)))               #Kaiser Rule
dim_est_CR[2] <- 1+length(which(cumsum(var_exp_CR)<=.85))                     #Variance Explained
dim_est_CR[3] <- which.max(pca_CR$sdev[2:col_num] / pca_CR$sdev[1:col_num-1]) #Ratio Rule
dim_est_CR[4] <- paran(CR)$Retained                                           #Horn's Parallel Analysis
dim_est_CR[5] <- mean(dim_est_CR[1:4])

#Dimension Estimation Plot
colnames(dim_est_CR) <- c("Kaiser", "Var. Exp.", "Ratio", "Horn's", "Average")
barplot(dim_est_CR, col = "skyblue2", xlab = "Method", ylab = "PC's Retained",
  main = "Number of PC's Retained")

#Loadings Plot
par(mar=c(5, 4, 4, 5), xpd=TRUE)
barplot(pca_CR$rotation^2, main = "PCA Loadings", col = c(1:13))
legend(x = "topright", inset = c(-0.1,0), legend = colnames(CR), col = c(1:13), pch = 15)

# Preliminary RPCA ---------------------------------------------------------

#RPCA
rpca_CR <- rpca(as.matrix(CR))
var_exp_CR2 <- rpca_CR$L.svd$d^2 / sum(rpca_CR$L.svd$d^2)
plot((1:length(var_exp_CR2)), var_exp_CR2, "l",main = "Scree Plot of Returns",
xlab = "Principal Components", ylab = "Variance Explained")
```

```r
#Dimension Estimation
dim_est_CR2 <- matrix(ncol = 5, nrow = 1)

dim_est_CR2[1] <- length(which(rpca_CR$L.svd$d > mean(rpca_CR$L.svd$d)))   #Kaiser Rule
dim_est_CR2[2] <- 1+length(which(cumsum(var_exp_CR2)<=.85))                #Variance Explained
dim_est_CR2[3] <- which.max(rpca_CR$L.svd$d[2:6] / rpca_CR$L.svd$d[1:6-1]) #Ratio Rule
dim_est_CR2[4] <- paran(x = rpca_CR$L)$Retained                             #Horn's Parallel Analysis
dim_est_CR2[5] <- mean(dim_est_CR2[1:4])

#Dimension Estimation Plot
colnames(dim_est_CR2) <- c("Kaiser", "Var. Exp.", "Ratio", "Horn's", "Average")
barplot(dim_est_CR2, col = "lightpink2", xlab = "Method", ylab = "PC's Retained",
main = "Number of PC's Retained")

#Loadings Plot
loadrpca <- t(rpca_CR$L.svd$vt)
colnames(loadrpca) <- sprintf("PC%d",seq(1:6))
par(mar=c(5, 4, 4, 5), xpd=TRUE)
barplot(loadrpca^2, main = "RPCA Loadings", col = c(1:13))
legend(x = "topright", inset = c(-0.1,0), legend = colnames(CR), col = c(1:13), pch = 15)

# WS Asset Analysis ------------------------------------------------------
#Creates Plots and Data Frames for analysis with EWS

for (i in 1:col_num){

  SumStat <- matrix(nrow = (row_num - win_size),ncol = 4)

  for (j in 1:(row_num - win_size)) {

    if (expanding == TRUE){
      j_0 = win_size
      main_title = "EWS Summary"
    } else {
      j_0 = j
      main_title = paste("RWS Summary with Window", win_size)
    }

    temp_set <- CR[j_0:(j + win_size), i]
    SumStat[j,1]<-mean(temp_set)
    SumStat[j,2]<-sd(temp_set)
    SumStat[j,3]<-skewness(temp_set)
    SumStat[j,4]<-kurtosis(temp_set)
  }

  dates <- seq(as.Date("2020-08-18"), by = "day", length.out = (row_num-win_size))
  SumStat <- data.frame(dates, SumStat)
```

```
  colnames(SumStat) <- c("Dates", "Mean", "Std. Dev.", "Skew", "Kurt")

  par(mfrow=c(2,2))
  plot(SumStat$Dates, SumStat$Mean, 'l', xlab = "Dates", ylab = "Mean")
  plot(SumStat$Dates, SumStat$`Std. Dev.`, 'l', xlab = "Dates", ylab = "Std. Dev")
  plot(SumStat$Dates, SumStat$Skew, 'l', xlab = "Dates", ylab = "Skew")
  plot(SumStat$Dates, SumStat$Kurt, 'l', xlab = "Dates", ylab = "Kurtosis")
  mtext(paste(colnames(CR)[i],main_title), side = 3, line = -19, outer = TRUE)
}

rm("i", "j", "j_0", "main_title", "temp_set", "SumStat", "dates")

# WS PCA ------------------------------------------------------------------

pca_WS <- matrix(nrow = (row_num - win_size), ncol = 13*3)
loading_1 <- matrix(nrow = (row_num - win_size), ncol = 13)
loading_2 <- matrix(nrow = (row_num - win_size), ncol = 13)
loading_3 <- matrix(nrow = (row_num - win_size), ncol = 13)
loading_4 <- matrix(nrow = (row_num - win_size), ncol = 13)
error <- matrix(nrow = (row_num - win_size), ncol = 1)

for (i in 1:(row_num - win_size)){

  if (expanding == TRUE){
    i_0 = 1
  } else {
    i_0 = i
  }

  temp_pca <- prcomp(CR[i_0:(win_size + i),])
  recon <- temp_pca$x[,1:4] %*% t(temp_pca$rotation[,1:4])
  error[i,1] <- (1/sqrt(win_size + i))*norm(as.matrix(CR)[1:(64+i),] - recon, "f")

  for(j in 1:col_num){
    pca_WS[i,j] <- temp_pca$sdev[j]
    pca_WS[i,(j + col_num)] <- (temp_pca$sdev[j])^2
  }
  pca_WS[i,14:26] <- pca_WS[i,14:26] / sum(pca_WS[i,14:26])
  loading_1[i,] <- as.matrix(t(temp_pca$rotation[,1]))
  loading_2[i,] <- as.matrix(t(temp_pca$rotation[,2]))
  loading_3[i,] <- as.matrix(t(temp_pca$rotation[,3]))
  loading_4[i,] <- as.matrix(t(temp_pca$rotation[,4]))
}

#Dimension Estimation
dim_est_pca <- matrix(nrow = nrow(pca_WS), ncol = 4)

for (i in 1:(nrow(pca_WS))){
```

```
  dim_est_pca[i,1] <- length(which(pca_WS[i,1:13] > mean(pca_WS[i,1:13])))#Kaiser Rule
  dim_est_pca[i,2] <- 1+length(which(cumsum(pca_WS[i,14:26])<.8)) #Variance Explained
  dim_est_pca[i,3] <- which.max(pca_WS[i,2:13] / pca_WS[i,1:12])  #Ratio Rule
  dim_est_pca[i,4] <- mean(dim_est_pca[i,1:3])                    #Average Dimension
}

#Loadings Plots
par(mfrow = c(2,2))
matplot(loading_1^2, ylab = "", type = "l", col = 1:13, main = "PCA Loading 1")
matplot(loading_2^2, ylab = "",type = "l", col = 1:13, main = "PCA Loading 2")
matplot(loading_3^2, ylab = "",type = "l", col = 1:13, main = "PCA Loading 3")
matplot(loading_4^2, ylab = "",type = "l", col = 1:13, main = "PCA Loading 4")
legend("topright", legend = colnames(CR), col=1:13, pch=1)

#Dimension Estimation Plots
par(mfrow = c(2,2))
plot.ts(dim_est_pca[,1], ylab = "PC's Retained", main = "Kaiser's Rule")
plot.ts(dim_est_pca[,2], ylab = "", main = "Variance Explained" )
plot.ts(dim_est_pca[,3], ylab = "PC's Retained", main = "Ratio Rule")
plot.ts(dim_est_pca[,4], ylab = "", main = "Average")

rm("i", "i_0", "j")

# WS RPCA ----------------------------------------------------------------

rpca_WS <- matrix(nrow = (row_num - win_size), ncol = 13*2)
loading_1 <- matrix(nrow = (row_num - win_size), ncol = 13)
loading_2 <- matrix(nrow = (row_num - win_size), ncol = 13)
errorrpca <- matrix(nrow = (row_num - win_size), ncol = 1)

for (i in 1:(row_num - win_size)){

  if (expanding == TRUE){
    i_0 = 1
  } else {
    i_0 = i
  }

  temp_rpca <- rpca(as.matrix(CR[i_0:(win_size + i),]))

  for(j in 1:7){
    rpca_WS[i,j] <- temp_rpca$L.svd$d[j]
    rpca_WS[i,j+7] <- (temp_rpca$L.svd$d[j])^2
  }
  rpca_WS[i,8:14] <- rpca_WS[i,8:14] / sum(rpca_WS[i,8:14])
  errorrpca[i,1] <- norm(temp_rpca$S, "F")
  loading_1[i,] <- abs(temp_rpca$L.svd$vt[1,])
  loading_2[i,] <- abs(temp_rpca$L.svd$vt[2,])
```

```
}

#Dimension Estimation
dim_est_rpca <- matrix(nrow = nrow(rpca_WS), ncol = 4)

for (i in 1:(nrow(rpca_WS))){
  dim_est_rpca[i,1] <- length(which(rpca_WS[i,1:7] > mean(rpca_WS[i,1:7]))) #Kaiser Rule
  dim_est_rpca[i,2] <- 1+length(which(cumsum(rpca_WS[i,8:14])<=.85)) #Variance Explained
  dim_est_rpca[i,3] <- which.max(rpca_WS[i,2:7] / rpca_WS[i,1:6])    #Ratio Rule
  dim_est_rpca[i,4] <- mean(dim_est_rpca[i,1:3])                     #Average Dimension
}

#Loadings Plots
par(mfrow = c(2,1))
matplot(loading_1, ylab = "", type = "l", col = 1:13, main = "RPCA Loading 1")
matplot(loading_2, ylab = "",type = "l", col = 1:13, main = "RPCA Loading 2")

#Dimension Estimation Plots
par(mfrow = c(2,2))
plot(dim_est_rpca[,1], type = "l", ylab = "PC's Retained", main = "Kaiser's Rule")
plot(dim_est_rpca[,2], type = "l",ylab = "", main = "Variance Explained" )
plot(dim_est_rpca[,3], type = "l",ylab = "PC's Retained", main = "Ratio Rule")
plot(dim_est_rpca[,4], type = "l",ylab = "", main = "Average")


rm("i", "i_0", "j")
```

B.2 Python Code

B.2.1 Autoencoder Code

```
import numpy as np
import pandas as pd
import os
import sys
import csv
import json
import xlrd
from datetime import datetime
from sklearn.preprocessing import scale
from tensorflow.keras.layers import Input, Dense, Lambda, Dropout
from tensorflow.keras.models import Model
from keras import regularizers
```

```
from keras.callbacks import Callback
from keras import backend as K
import tensorflow as tf
from keras.regularizers import Regularizer
import scipy.sparse
from keras.models import load_model
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from sklearn.model_selection import GridSearchCV
from keras.wrappers.scikit_learn import KerasClassifier
from tensorflow import keras
from scipy.linalg import svdvals
import openpyxl
tf.compat.v1.get_default_graph()
tf.compat.v1.disable_v2_behavior()
tf.compat.v1.disable_eager_execution()

#Load in the return data and format the date column
crypt_ret = pd.read_excel(r"C:\Users\gigak\Downloads\APIData_Updated_SET_LOGReturns.xlsx")
crypt_ret['time'] = pd.to_datetime(crypt_ret['time'], format ='%Y -%m -%d')
#The set index pandas function here modifies the existing data frame, and the first parameter
#of the function uses one or more existing columns or arrays (of the correct length)
crypt_ret.set_index(crypt_ret['time'], inplace = True)
crypt_ret.drop(columns = ['time'], inplace = True)


#This function definition scales the return data between -1 and 1 as well as separatese the data between
#training and testing
def get_ret_data(k,h, crypt_ret = crypt_ret):
        X = crypt_ret.values
        X = X.astype('float32')
        X = X[0: (890 + h), :]
        #X = scale(X)
        X = X / np.max(np.abs(X))
        #x_train = X[0: (712 + k), :]
        #x_test = X[(713 + k):(890+k+h),:]

        #x_train = X[0: 890, :]
        #x_test = [[]]

        x_train = X[0: (63 + k), :]
        x_test = X[0: 890, :]
        return x_train, x_test, crypt_ret


#This function specifies the loss function for our autoencoder model
def mse_l1_loss(encoded_layer, lambda_):
    def loss(y_true, y_pred):
```

```
        return K.mean(K.square(y_pred - y_true) + lambda_*K.sum(K.abs(encoded_layer)))
    return loss


#This function builds the autoencoder architecture. The function is currently set up for a SLAE
#but can be generalized to include more layers.
def build_l1_ae_model(l1_reg, input_dim, encoding_dim, activation1, activation2, lr):
    input_img = tf.keras.Input(shape=(input_dim))
    z_layer_input = Lambda(lambda x:K.l2_normalize(x,axis=1))(input_img)
    encoded = tf.keras.layers.Dense(encoding_dim, activation=activation1)(z_layer_input)
    encoded_norm = Lambda(lambda x:K.l2_normalize(x,axis=1))(encoded)

    #create encoder model
    encoder = Model(input_img, encoded)
    decoded = Dense(input_dim, activation=activation2)(encoded)

    #create autoencoder model
    autoencoder = Model(input_img, decoded)

    opt = keras.optimizers.Adam(learning_rate = lr)
    autoencoder.compile(optimizer=opt, loss = mse_l1_loss(encoded_norm, l1_reg))
    return encoder, autoencoder


#This function builds the autoencoder architecture. The function is currently set up for a MLAE.
def build_mlae_model(l1_reg, input_dim, z1_dim, encoding_dim, z2_dim, activation_z1,
    activation_mid, activation_z2, activation_dec, lr):
    input_img = tf.keras.Input(shape=(input_dim))
    z1_layer_input = Lambda(lambda x:K.l2_normalize(x,axis=1))(input_img)
    z1_layer = tf.keras.layers.Dense(z1_dim, activation = activation_z1)(z1_layer_input)
    z1_layer_norm = Lambda(lambda x:K.l2_normalize(x,axis=1))(z1_layer)

    mid_layer_input = z1_layer_norm
    encoded = tf.keras.layers.Dense(encoding_dim, activation=activation_mid)(mid_layer_input)
    encoded_norm = Lambda(lambda x:K.l2_normalize(x,axis=1))(encoded)

    #create encoder model
    encoder = Model(input_img, encoded)

    z2_layer_input = encoded_norm
    z2_layer = tf.keras.layers.Dense(z2_dim, activation = activation_z2)(z2_layer_input)
    z2_layer_norm = Lambda(lambda x:K.l2_normalize(x,axis=1))(z2_layer)
    decoded = Dense(input_dim, activation=activation_dec)(z2_layer_norm)

    #create autoencoder model
    autoencoder = Model(input_img, decoded)

    opt = keras.optimizers.Adam(learning_rate = lr)
```

```
    autoencoder.compile(optimizer=opt, loss = mse_l1_loss(encoded_norm, l1_reg))
    return encoder, autoencoder


#This function sorts the rows of a data matrix independently from largest to smallest values
def sort_by_row(z):
    z_sorted = None
    for i in np.arange(z.shape[0]):
        #sorted in reverse yields a data matrix in descending order
        z_s = sorted(z[i,:], reverse=True)
        if z_sorted is None:
            z_sorted = z_s
        else:
            #This vstack command builds the sorted matrix up iteratively by first finding
            #the next row to be appended and then appending it on the existing matrix.
            #It begins with an empty data matrix.
            z_sorted = np.vstack((z_sorted, z_s))
    return z_sorted


#This function estimates the dimension of a data matrix by taking
#the singular values or SVPs and computing how many SVPs explain
#more than the specified threshold percentage variance
def algorithm_2(z, threshold):
    tot = sum(z)
    z_pct = [(i/tot) for i in sorted(z, reverse = True)]
    z_gt_theta = [i for i in z_pct if i >= threshold]
    return len(z_gt_theta)


#This function estimates the dimension of a matrix by taking the ordered singular values
#or SVPs and sequentially finding out how many values are needed to explain the
#threshold cumulative percent variance
def algorithm_3(SVPs, threshold):
    tot = sum(np.square(SVPs))
    dim = 0
    for i in range(1, (len(SVPs) + 1)):
        if sum(np.square(SVPs[0:i]))/tot >= threshold:
            dim = i
            return dim


#This code builds the autoencoder model based off one combination parameter choices
#and fits it to the data for one k value
activation_z1 = 'elu'
activation_z2 = 'elu'
activation_mid = 'elu'
activation_dec = 'sigmoid'
```

```
activation1 = 'softmax'
activation2 = 'tanh'
lr = 0.001
l1_reg = 5e-5
z1_dim = 12
encoding_dim = 12
z2_dim = 12
epochs = 1000
batch_size = 256
k= [1, 96, 97, 98, 99, 100, 306, 307, 308, 309, 310, 566, 567, 568, 569, 570, 826]
KVDE_list = []
svps = []
N = 5
avg_loss_total = 0
for j in k:
    z_mus = []
    k_val_dim_ests = []
    for q in range(0,N):
        x_train, x_test, crypt_ret = get_ret_data(k=j, h = 0)
        input_dim = x_train.shape[1]
        encoder, autoencoder = build_l1_ae_model(l1_reg = l1_reg, input_dim = input_dim,
        encoding_dim = encoding_dim, activation1 = activation1,
        activation2 = activation2, lr = lr)
        #encoder2, autoencoder2 = build_mlae_model(l1_reg = l1_reg, input_dim = input_dim,
        #z1_dim = z1_dim, encoding_dim = encoding_dim, z2_dim = z2_dim, activation_z1 = activation_z1,
        #activation_mid = activation_mid, activation_z2 = activation_z2, activation_dec = activation_dec,
        #lr = lr)
        history1 = autoencoder.fit(x_train, x_train, epochs = epochs, batch_size = batch_size, verbose = 0)
        total_loss = 0
        for i in history1.history['loss']:
            total_loss += i
        print(f"Average loss at step {j} is {(total_loss / len(history1.history['loss']))}")
        avg_loss_total += (total_loss / len(history1.history['loss']))
        #history2 = autoencoder2.fit(x_train, x_train, epochs = epochs, batch_size = batch_size, verbose = 0)

        #This code computes the SVPs and predicts the estimated dimension using the
        #algorithm_2() and algorithm_3() functions
        z1 = encoder.predict(x_train)
        z_row_sorted1 = sort_by_row(z1)
        #SVPs
        z_mu1 = np.mean(z_row_sorted1, axis = 0)
        alg_2_dim1 = algorithm_2(z_mu1, 0.01)
        alg_3_dim1 = algorithm_3(z_mu1, 0.85)

        #z2 = encoder2.predict(x_train)
        #z_row_sorted2 = sort_by_row(z2)
        #SVPs
        #z_mu2 = np.mean(z_row_sorted2, axis = 0)
```

```
        #alg_2_dim2 = algorithm_2(z_mu2, 0.01)
        #alg_3_dim2 = algorithm_3(z_mu2, 0.85)

        #print(alg_2_dim)
        print(f"Estimated dimension at step k = {j} is: {alg_3_dim1}")
        k_val_dim_ests.append(alg_3_dim1)
        z_mus.append(z_mu1)
        #print(z_mus)
        #print('Estimated dimension for deep autoencoder:', alg_3_dim2)
        #print(tf.version.VERSION)
        #print(z_mu2)
        #print(z2)
        #print(z_row_sorted)

    #with open('autoencoder_recon.csv', 'w', newline='') as file:
        #writer = csv.writer(file)
        #writer.writerows(z)
    z_mu_avg = [sum(sub_list) / len(sub_list) for sub_list in zip(*z_mus)]
    svps.append(z_mu_avg)
    KVDE_list.append(k_val_dim_ests)

print(svps)
avg_loss_total = avg_loss_total / (N * len(k))
print("The average total loss over all trials is: ", avg_loss_total)

final_dimension_estimates = []
for x in KVDE_list:
    final_dimension_estimates.append(sum(x) / len(x))

print(final_dimension_estimates)

z_scree = []
for j in svps:
    individual_scree = []
    for i in range(len(j)):
        individual_scree.append(j[i] / sum(j))
    z_scree.append(individual_scree)

plt.rcParams['figure.figsize'] = (8,8)
for i in range(len(z_scree)):
    x_axis = [1,2,3,4,5,6,7,8,9,10,11,12]
    y_axis = z_scree[i]
    plt.plot(x_axis, y_axis, label = "k = {value}".format(value = k[i]))
    plt.legend()

plt.xlabel('SVP Number')
plt.ylabel('Variance Explained')
plt.show()
```

B.2.2 Binance API Code

```
# import libraries
from binance.spot import Spot as Client
import pandas as pd
import plotly.graph_objects as go
from IPython.display import display

# url to access Binance API
base_url = "https://api.binance.com"

# create Client to access API
spot_client = Client(base_url = base_url)

# requesting exchange info
# used to access list of assets and their permission
exchange_info = spot_client.exchange_info()
exchange_info

# access historical prices
btcusd_history = spot_client.klines("BTCUSDT", "1d", limit = 50000)
display(btcusd_history[:2])

# constructs dataset
columns = ['time', 'open', 'high', 'low', 'close', 'volume', 'close_time',
'quote_asset_volume', 'number_of_trades', 'taker_buy_base_asset_volume',
'taker_buy_quote_asset_volume', 'ignore']

# writes data into csv file
with open('test.csv', 'w') as f:
    writer = csv.writer(f)
    for i in range(len(btcusd_history)):
        writer.writerow(btcusd_history[i])
```

Appendix C

EDA Plots

C.1 Prices

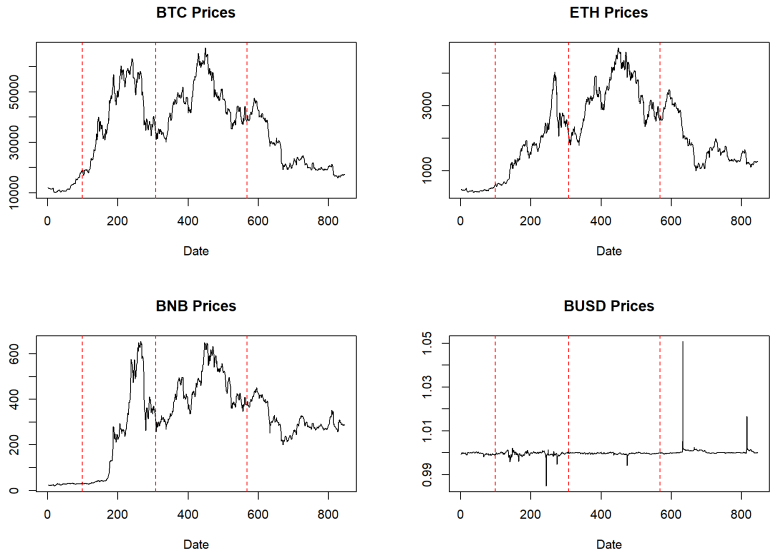


Figure C.1: Price of BTC, ETH, BNB and BUSD plotted over time.

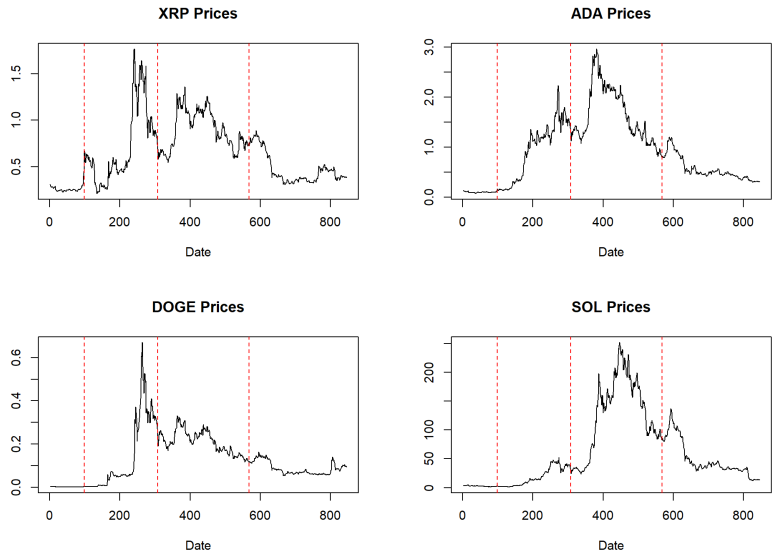


Figure C.2: Price of XRP, ADA, DOGE and SOL plotted over time.

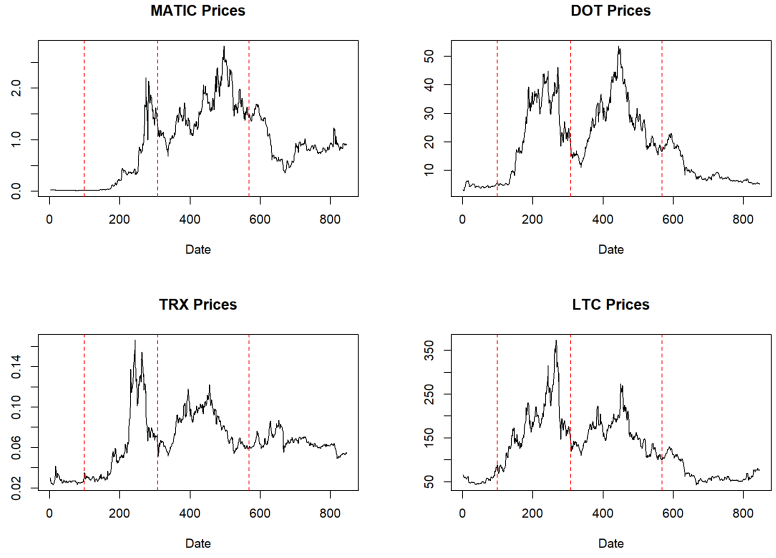


Figure C.3: Price of MATIC, DOT, TRX and LTX plotted over time.

C.2 Returns

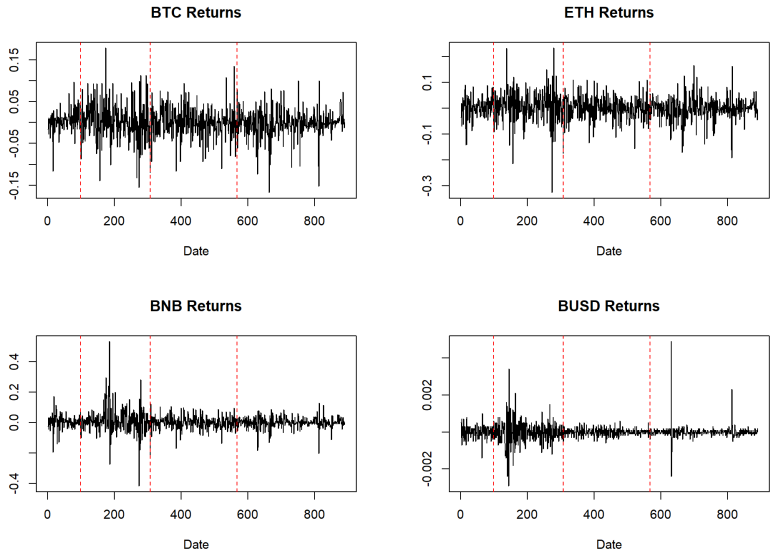


Figure C.4: Returns of BTC, ETH, BNB and BUSD plotted over time

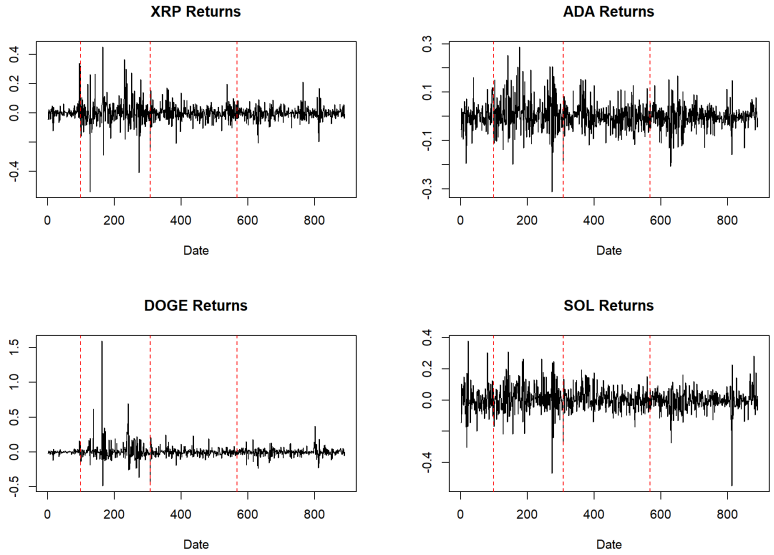


Figure C.5: Returns of XRP, ADA, DOGE and SOL plotted over time.

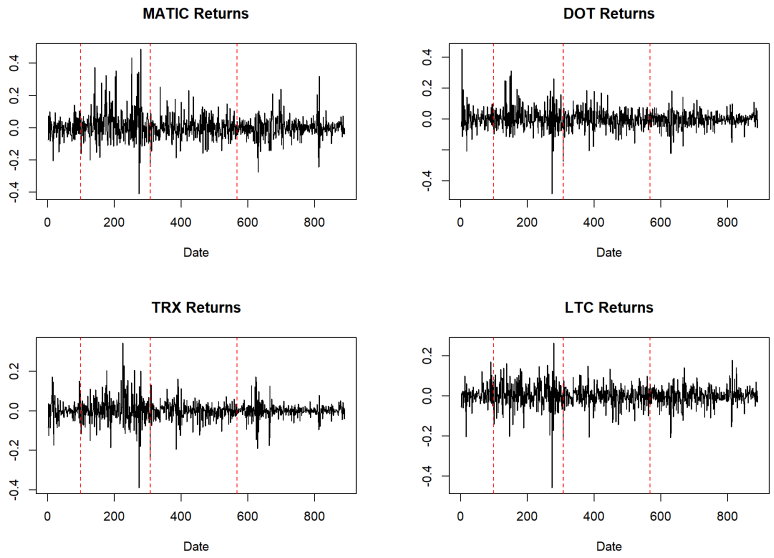


Figure C.6: Returns of MATIC, DOT, TRX and LTX plotted over time.

C.3 Expanding Window Scheme

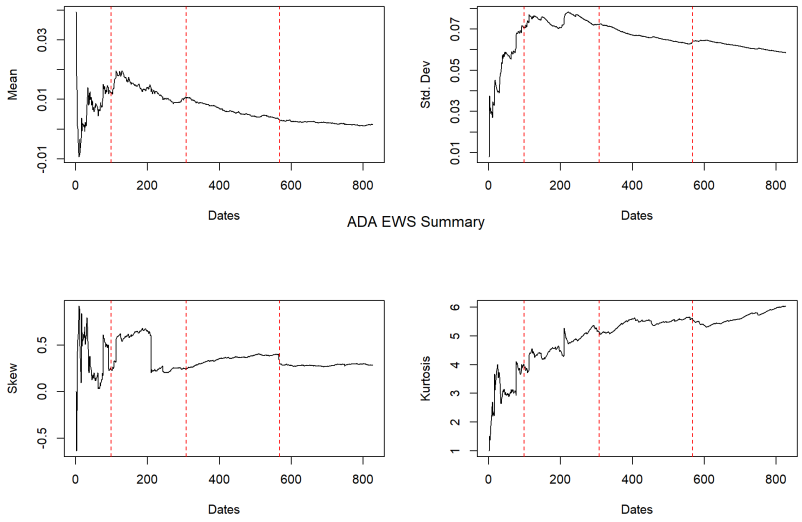


Figure C.7: EWS summary statistics of ADA.

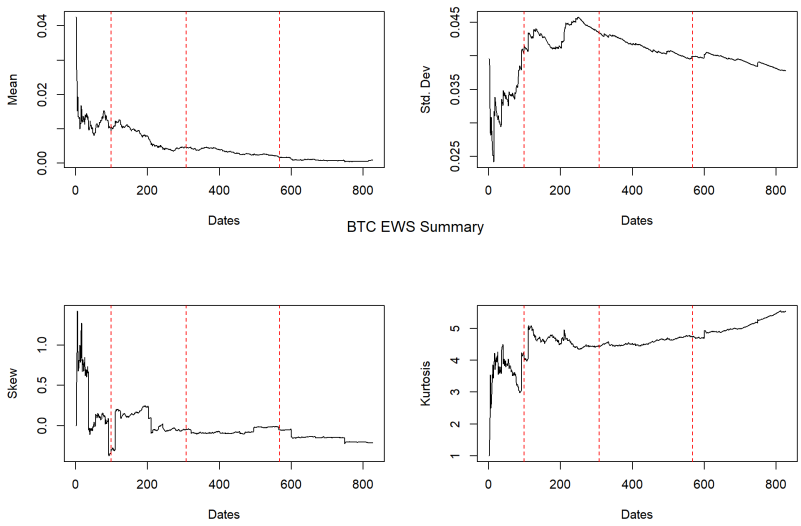


Figure C.9: EWS summary statistics of BTC.

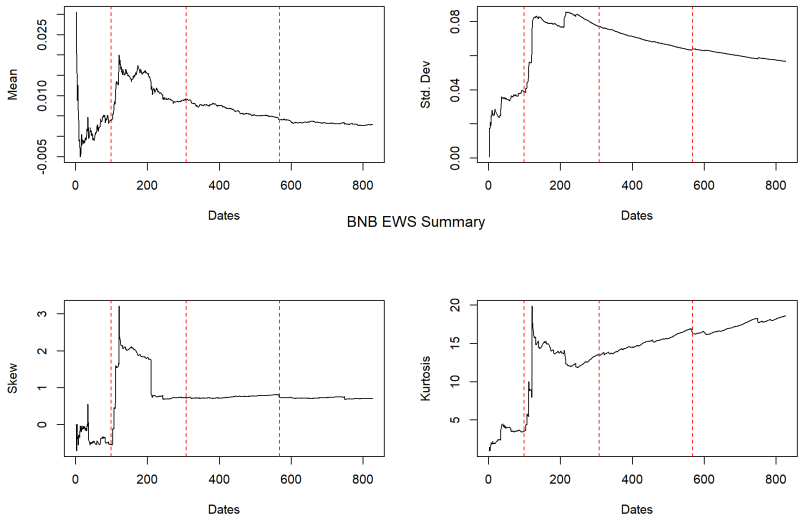


Figure C.8: EWS summary statistics of BNB.

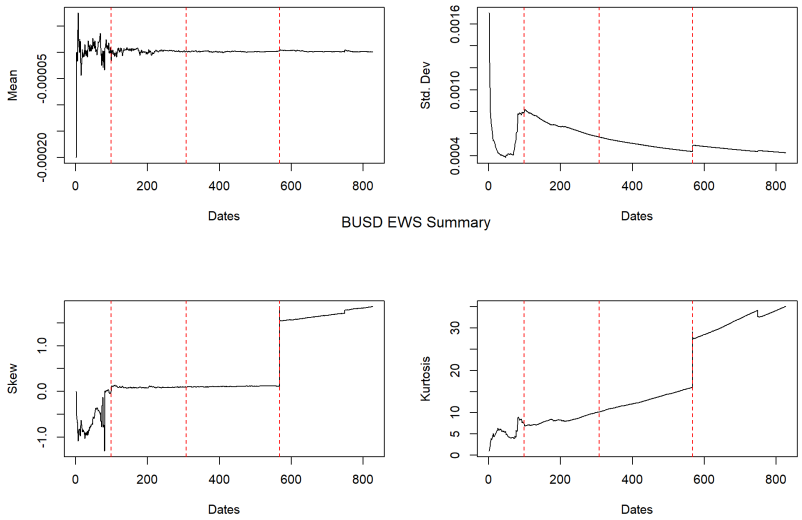


Figure C.10: EWS summary statistics of BUSD.

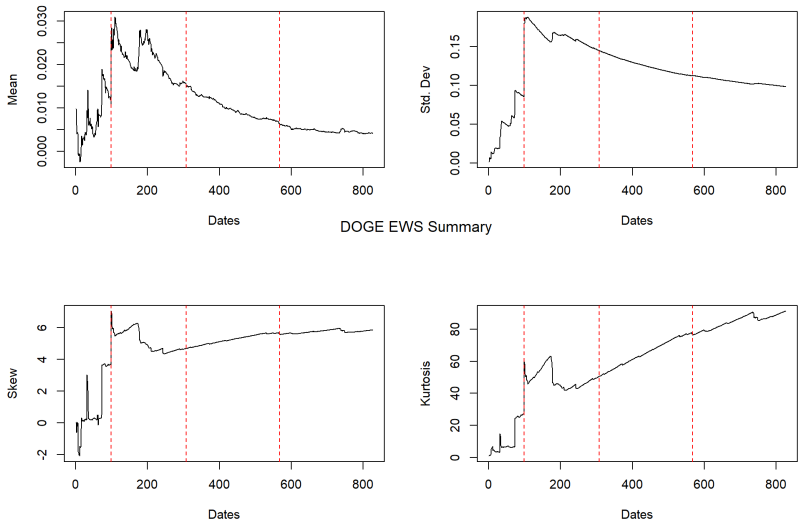


Figure C.11: EWS summary statistics of DOGE.

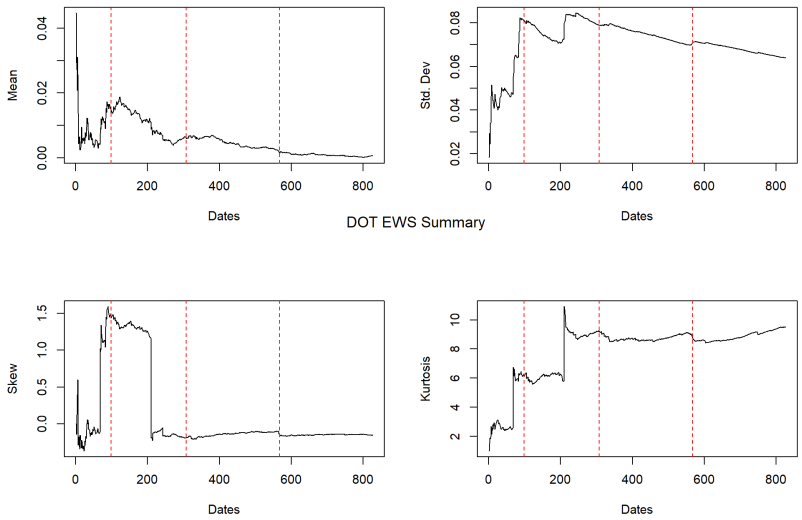


Figure C.12: EWS summary statistics of DOT.

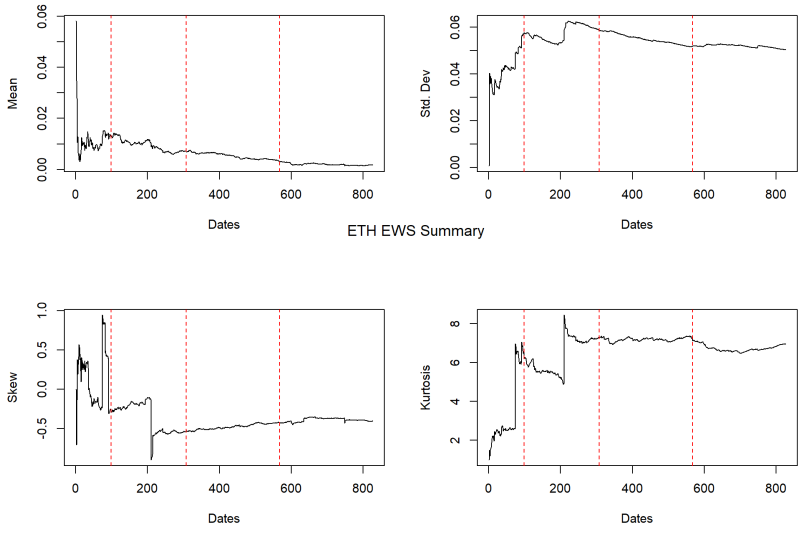


Figure C.13: EWS summary statistics of ETH.

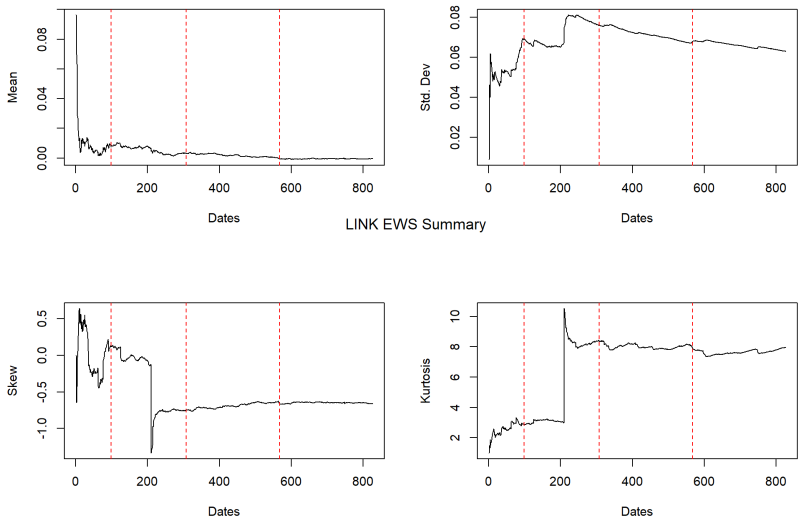


Figure C.14: EWS summary statistics of LINK.

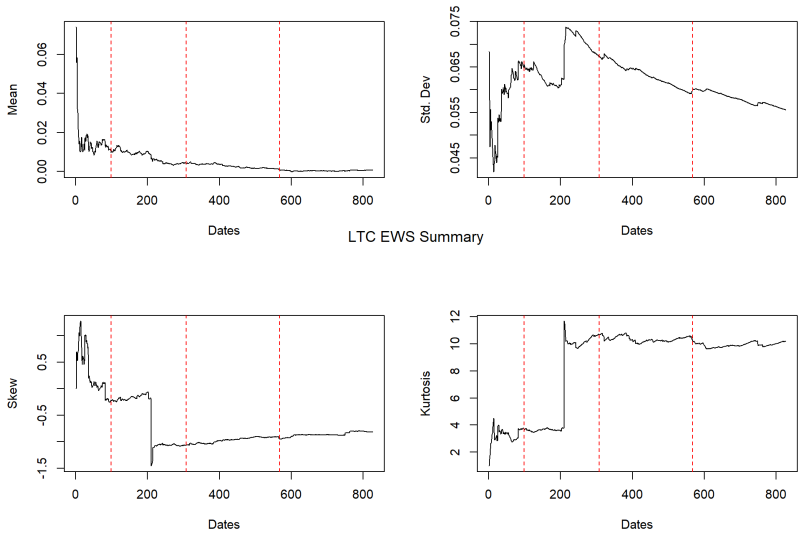


Figure C.15: EWS summary statistics of LTC.

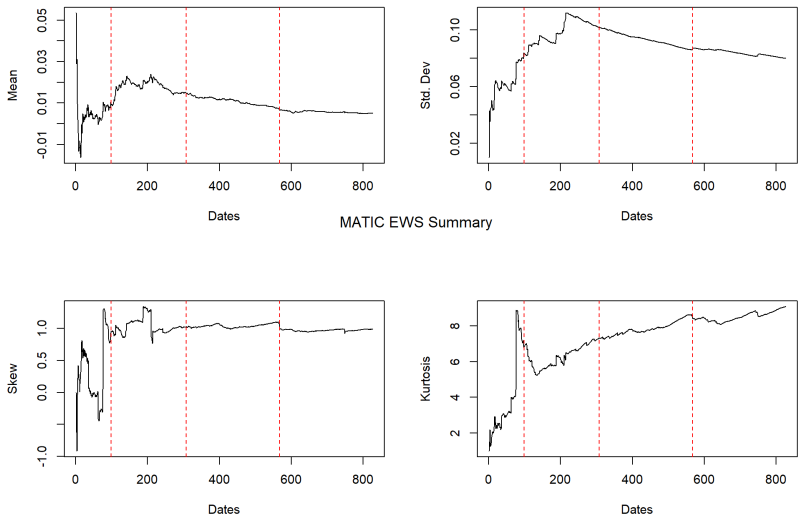


Figure C.16: EWS summary statistics of MATIC.

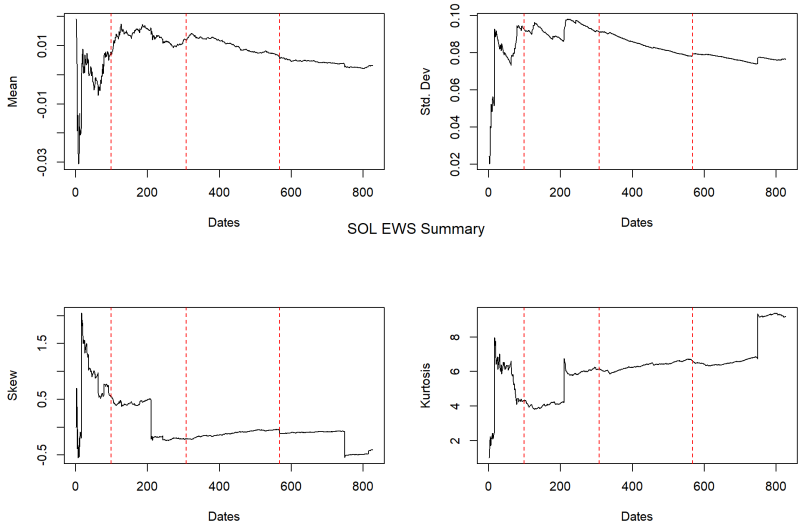


Figure C.17: EWS summary statistics of SOL.

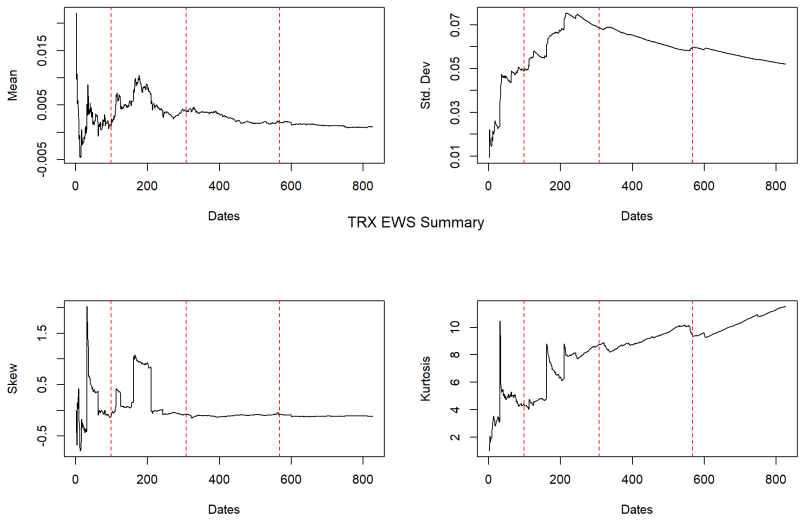


Figure C.18: EWS summary statistics of TRX.

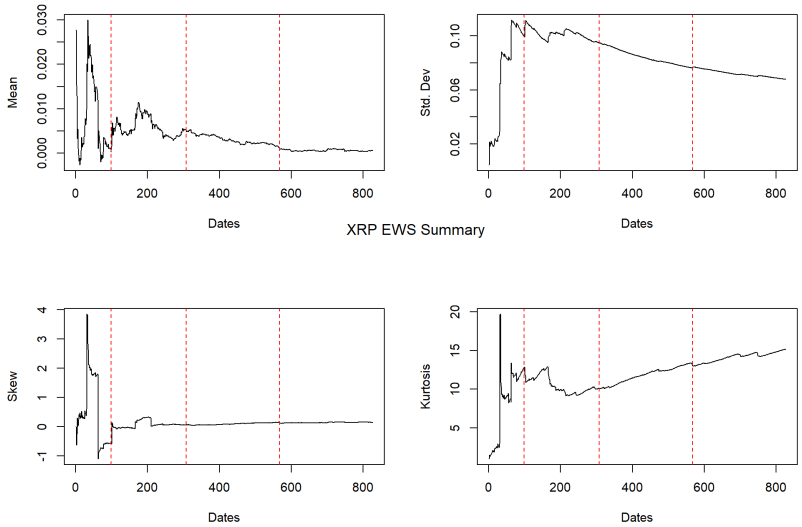


Figure C.19: EWS summary statistics of XRP.