CRYPTOGRAPHIC ALGORITHMS, CRYPTOCURRENCIES, AND A PREDICTIVE MODEL OF BITCOIN VALUE BY PLS REGRESSION

by

PAUL KENNETH O'CONNOR

A THESIS

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

APPLIED MATHEMATICS

2024

Approved by:

Wenqing Hu, Advisor
Akim Adekpedjou
Gayla R. Olbricht

ABSTRACT

With the invention of Bitcoin in 2009, as a seemingly timed response to the ongoing financial crisis, the popularity of the cryptocurrency has since continued to grow. Just this year, the Security Exchange Commission approved Bitcoin for exchange traded funds, allowing major investment firms to begin product trading. With this approval, and during this very moment of writing, Bitcoin has entered a bull market and reached a record value of over 72,000 USD. In addition, the Bitcoin halving event in April of 2024 is expected to increase demand even further. It has been anticipated that Bitcoin and other cryptocurrencies will continue to grow in popularity as an alternate source of investment. The purpose of this thesis is to review two important cryptographic algorithms currently being implemented to validate and trade cryptocurrencies, SHA-3 and ECDSA. Furthermore, we provide a general overview of how Bitcoin, Ethereum, and their networks operate independently of third parties. Finally, a successful analysis using Partial Least Squares regression is conducted to predicted the moving average price of Bitcoin with hope the methodology could be implemented in future trading strategies.

ACKNOWLEDGMENTS

We would like to acknowledge and thank Dr. Wenqing Hu for his continual support and encouragement. We would also like to thank additional members of the advisory committee, Dr. Akim Adekpedjou and Dr. Gayla Olbricht. Finally, a special thanks to Luozhu and Yicheng, who were willing to meet and discuss their expertise in blockchain and crypto trading.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

The review of cryptographic algorithms beings with an overview of the Secure Hash Algorithm three (SHA-3), a series of algorithms used to encode a message and the primary algorithm used to ensure participants using a cryptocurrency remain anonymous. The review of this algorithms specifies how a string of bits can be reoriented to entirely different sequence of bits. Another important algorithm used in conjunction with the SHA-3 is sponge construction. This algorithm takes multiple outputs from SHA-3 and compresses them into a predetermined length, allowing encrypted messages to be sent with minimal use of memory. Within blockchain technology, the Merkle tree provides the structure to store the output from the SHA-3 and sponge construction algorithms. Finally, the Elliptical Curve Digital Signature Algorithm (ECDSA) is reviewed to provide an overview of how participants verify signatures to ensure transactions are being received from the intended individual.

Following the review of current cryptographic algorithms in use, we discuss the basic concepts governing the trading of Bitcoin and its founding principles. Of which, the most important concept was the resolution of the double spending problem, negating the need for third party verification systems. After the review of Bitcoin, we provide a review of Ethereum, the Virtual Machine, and its use of smart contracts. Smart contracts have become increasingly important through out the development of cryptocurrencies and allow participants to code functions for increased versatility in trading and exchanging different cryptocurrencies.

Finally, on-chain data from Bitcoin was obtained to generate a model to predict the moving average price of Bitcoin using Partial Least Squares (PLS) regression. The time period of analysis incorporated the 2020 Bitcoin halving event, which reduced the reward a miner can receive for validating a transaction by one half. This time period provides an opportunity to develop a model for price prediction during high volatility. It is our hope that development of such a model may lead to further development of predictive models implemented in new trading techniques for crypto-assets.

2. CRYPTOGRAPHIC ALGORITHMS USED IN BLOCKCHAIN

2.1. SECURE HASH ALGORITHM

In 2015, the National Institute of Standards and Technology selected the KECCAK algorithm as the standardized Secure Hash Algorithm-3 (SHA-3) for cryptographic hash algorithms. The algorithm is comprised of four cryptographic hash functions and two extendable-output functions (XOFs). The four cryptographic hash functions are referred to as SHA3-224, SHA3-256, SHA3-384, and SHA3-512, each of which produce a bit hash value equivalent in length to the number specified in the nomenclature. The remaining two XOFs are referred to as SHAKE128 and SHAKE256, used to extend the output beyond the specified value [1]. One of the primarily applications of the hash functions is to encrypt transactional data stored within block-chain technology.

All six functions have a shared structure known as sponge construction. Each of the six SHA-3 functions utilize the same underlying algorithms to generate permutations within the sponge construction. These families of permutations are collectively known as KECCAK-*p* permutations [1].

The following sections discuss the algorithms implemented in KECCAK-*p* permutations and their application in sponge construction. First, the algorithms of KECCAK-*p* will be addressed followed by a discussion of sponge construction.

2.1.1. KECCAK-*p* Permutations. KECCAK-*p* permutations have two parameters. The first parameter is the length of the string intended to be permuted, denoted by b, and called the *width* of the permutation. The second parameter is the number of rounds the string undergoes, denoted by n_r. Each round consists of five algorithmic transformations known as step mappings. Overall, the permutation is expressed by KECCAK-$p[b, n_r]$, with the width taking any of the following values; 25, 50, 100, 200, 400, 800, 1600. Before the string is taken through each algorithmic step mapping, it can be conceptualized as a reoriented three-dimensional state array [1].

2.1.2. State Array. Each permutation of KECCAK-$p[b,\ n_r]$ consists of b bits. While the input and output are represented by strings, the internal state undergoing transformations is represented by a three dimensional 5-by-5-by-w array, known as the *state array*. The variable w is equal to the permutation width, b, divided by twenty-five. Another variable used in discussion of the KECCAK-$p[b,\ n_r]$ permutations is the binary logarithm l, calculated $\log_2(b/25)$. A single bit within the state array located at a coordinates $(x,\ y,\ z)$ is denoted as $\mathbf{A}[x,\ y,\ z]$ [1]. Figure 2.1 displays the representation of a three dimensional 5-by-5-by-8 state array with a width of 200. The $x,\ y,$ and z coordinates of the state array are labeled along the edge with coordinates $\mathbf{A}[0,0,0]$ representing the central square of the first slice.

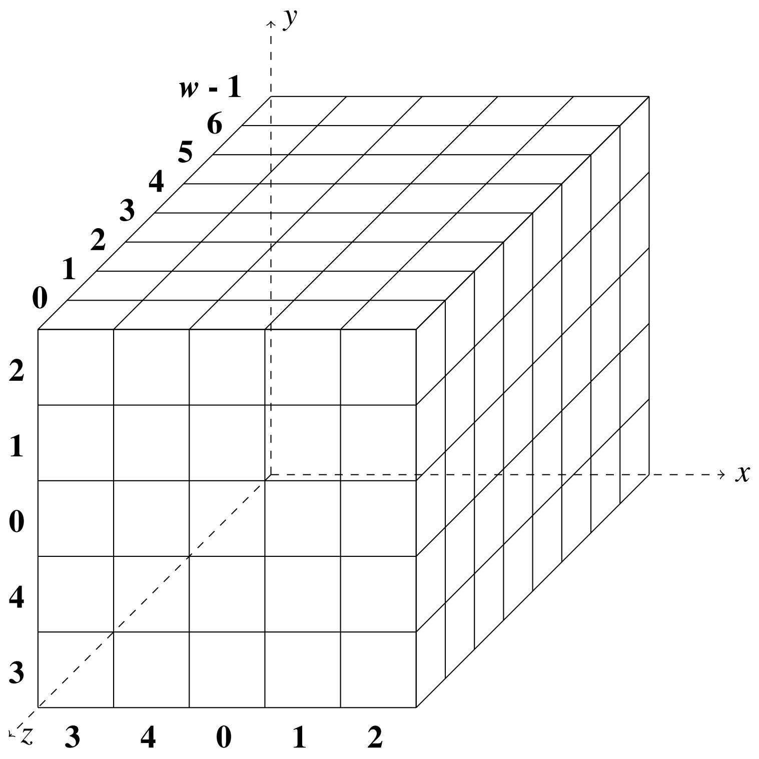


Figure 2.1. A 5-by-5-by-8 state array with a width of 200.

The length of bits along a z axis are known as *lanes*, while the x and y directions are known as a *rows* and *columns* respectively. Two-dimensional planes spanning the entire axes x and y, z and x, and z and y are known as *slice*, *plane*, and *sheet* respectively. Converting a string of b bits, where each individual bit is denoted by S[0], S[1], S[2], . . . , S[b-1], into a state array is defined by equation 2.1 for all triplets (x, y, z) such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$. It should be noted the previously mentioned criteria for all triplets (x, y, z) is required for all step mappings.

$$\mathbf{A}[x, y, z] = S[w(5(y + x) + z] \tag{2.1}$$

Once the state array has gone through the specified number of rounds in KECCAK-$p[b, n_r]$, the output is again represented by a string of bits. Each bit within a lane can be concatenated to the neighboring bit starting with the coordinates **A**[0, 0, 0]. The resulting string from a lane is then concatenated to the next lane, until all bits are transformed into one string. For example, the *Lane* (0,0) will be concatenated to *Lane* (1,0), and this result will then be concatenated to *Lane* (2,0). Equation (2.2) expresses the concatenation of all bits within a lane. Concatenating neighboring lanes, will from a plane, and subsequently the concatenation of the planes will yield the string of bits, as expressed in equations 2.3 and 2.4. [1].

$$Lane(i, j) = \mathbf{A}[i,j,0] \parallel \mathbf{A}[i,j,1] \parallel \mathbf{A}[i,j,2] \parallel ... \parallel \mathbf{A}[i,j,w-1] \tag{2.2}$$

$$Plane(j) = Lane(0, j) \parallel Lane(1, j) \parallel Lane(2, j) \parallel Lane(3, j) \parallel Lane(4, j) \tag{2.3}$$

$$S = Plane(0) \parallel Plane(1) \parallel Plane(2) \parallel Plane(3) \parallel Plane(4) \tag{2.4}$$

2.1.3. Overview of Step Mappings. One round of permutations within KECCAK-$p[b, n_r]$ consists of five algorithms collectively known as step mappings, denoted by θ, ρ, π, χ, and ι. Each step begins with the state array **A** as the input and returns the transformed state array **A**' as the output, such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$. Within each round the steps are applied iteratively with the output from a preceding round becoming the input for the next. The following sections describe each of the five algorithms of the step mappings [1].

2.1.4. Algorithm θ(A). The steps to compute $\theta(\mathbf{A})$ are defined by equations 2.5 through 2.7. The first step C[x, z] sequentially applies an exclusive OR function (XOR) to each bit within a column, beginning with the bit located at $y = 0$ and ending at $y = 4$. The next step D[x, z] specifies C[x, z] to be computed on the columns [(x-1) mod 5, z] and [(x+1) mod 5, (z-1) mod w], relative to the column **A**[x, z]. The results of the XOR on the two columns are subsequently XORed against one another. Finally, the bit within **A**[x, y, z] is XORed against D[x, z] to derive its new value within **A'**[x, y, z]. These steps are applied to each bit within the state array [1].

$$C[x,\ z] = \mathbf{A}[x, 0, z] \oplus \mathbf{A}[x, 1, z] \oplus \mathbf{A}[x, 2, z] \oplus \mathbf{A}[x, 3, z] \oplus \mathbf{A}[x, 4, z] \qquad (2.5)$$

$$D[x,\ z] = C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod w] \qquad (2.6)$$

$$\mathbf{A'}[x,\ y,\ z] = \mathbf{A}[x,\ y,\ z] \oplus D[x,z] \qquad (2.7)$$

2.1.5. Algorithm ρ(A). The steps to compute $\rho(\mathbf{A})$ are defined by equations 2.8 through 2.11. The purpose of $\rho(\mathbf{A})$ is to rotate each bit within its lane to a new position, called the offset. The value t is seeded prior to the execution of KECCAK-$p[b,\ n_r]$ and designates all lanes in the state array zero through twenty-three, with the exception of *lane* (0, 0). Step 2.11 specifies that the bit in position z is to be rotated by adding the offset along the direction of the z-axis, modulo the length of the lane. Along with the rotation by the offset, the entire lane is repositioned to the new lane **A**(y, (2x + 3y) mod 5), relative to the initial lane **A**(x, y). As shown in 2.8, lane **A**[0, 0] remains unchanged. Executing steps 2.8

through 2.11 returns the permuted state array **A’** [1].

$$\text{Let } \mathbf{A'}[0,0,z] = \mathbf{A}[0,0,z] \tag{2.8}$$

$$\text{Let } (x,y) = (0,1) \tag{2.9}$$

$$\text{For } t \text{ from 0 to 23: let } \mathbf{A'}[x,y,z] = \mathbf{A}[x,y,(z+(t+1)(t+2)/2) \bmod w] \tag{2.10}$$

$$\text{Let } (x,y) = (y,(2x+3y) \bmod 5) \tag{2.11}$$

2.1.6. Algorithm π(A). The algorithm to compute π(**A**) is defined by equation 2.12. The purpose of this step is to rearrange the position of the lanes within the state array. While the bit values along the length of a lane remain unchanged, their new position at coordinate *y* is determined by the initial position of *x*. In other words, an initial row of *x* values is transposed into a column of *y* values, and is applied to the entirety of a lane. Finally, the *x* value is determined by [(*x* + 3*y*) mod 5]. This step is carried out for every lane in the state array [1].

$$\mathbf{A'}[x,y] = \mathbf{A}[(x+3y) \bmod 5, x] \tag{2.12}$$

2.1.7. Algorithm χ(A). The algorithm to compute χ(**A**) is defined by equation 2.13. χ(**A**) uses a nonlinear function to XOR a bit within the state array against the next two bits within the same row. Relative to the bit **A**[*x, y, z*], the bit located at **A**[(*x*+1) mod 5, *y*, *z*] first undergoes and an XOR with the value one. The result of this XOR then undergoes the Boolean AND operation with the bit **A**[(x+2) mod 5, y, z]. Finally, the product of the Boolean AND operation is XORed with the bit **A**[*x, y, z*]. This operation is conducted for every bit within the state array [1].

$$\mathbf{A'}[x,y,z] = \mathbf{A}[x,y,z] \oplus ((\mathbf{A}[(x+1)] \bmod 5, y, z] \oplus 1) \cdot \mathbf{A}[(x+2) \bmod 5, y, z]) \tag{2.13}$$

2.1.8. Algorithm $\iota(\mathbf{A}, i_r)$. The round index i_r is a parameter for $\iota(\mathbf{A}, i_r)$ where the value of the round index coincides with the current round being executed, and can fall between 0 to (n_r - 1). The effect of $\iota(\mathbf{A}, i_r)$ modifies some of the bits within the lane **A**[0, 0] while other lanes remain unchanged. Equation 2.14 shows the round constant XORed with the central lane [1]. For simplicity, the round constant $RC[k]$ can be expressed as a hexadecimal, displayed in table 2.1 for twenty-four rounds in KECCAK-*p*[1600, 24]. The hexadecimal represents a binary based number system. For every hexadecimal, the resulting number can be expressed as an integer equal to the offset modulo the lane size.

$$\text{Let } \mathbf{A'}[0, 0, z] = \mathbf{A}[0, 0, z] \oplus RC[k] \tag{2.14}$$

Table 2.1. Round constant $RC[k]$ with the associated hexadecimal for twenty-four rounds

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

2.1.9. Summary and KECCAK-f[b]. The round function, denoted as *Rnd*, operates on the state array **A** and round index i_r. The round function designates the order of each step mapping the state array undergoes, with the output of each step becoming the input for the next. The entire round function Rnd(**A**, i_r), including all step mappings associated with one round is expressed in equation 2.15 [1].

The entirety of KECCAK-p[b, n_r], starting with the unencrypted string, can be summarized with the following steps. First, a string is converted into a state array. Next, the round index i_r can take any value values between (12+2l-n_r) to (12+2l-1). Finally, the result is converted back into a string using sponge construction, which will be addressed further in the following section [1].

It should be noted there is specialized version of KECCAK-p[b, n_r] known as KECCAK-f[b]. The number of rounds, n_r, is omitted from the notation given it is always equal to (12 + 2l). For example, when l = 6, KECCAK-p[1600, 24] is equal to KECCAK-f[1600]. It is this version of KECCAK-f[b] that is used when encrypting transactions in electronic currency [1].

$$Rnd(\mathbf{A}, i_r) = \iota(\chi(\pi(\theta(\mathbf{A}))), i_r) \tag{2.15}$$

2.2. SPONGE CONSTRUCTION

Sponge construction is the application of the cryptographic step mappings on a string to yield a single and final encrypted string. There are three components that comprise sponge construction; the padding rule, the parameter known as the *rate*, and the underlying function of KECCAK-p[b, n_r] which is executed on a fixed string length denoted as f. The sponge function is denoted by SPONGE[f, *pad*, r]. Figure 2.2 displays the general diagram of sponge construction [1].

2.2.1. Padding Rule. The padding rule takes the initial string N, otherwise known as the message to be encrypted, and concatenates a separate string of a required length to yield a resultant string that is a multiple of the rate. The padding rule is denoted as

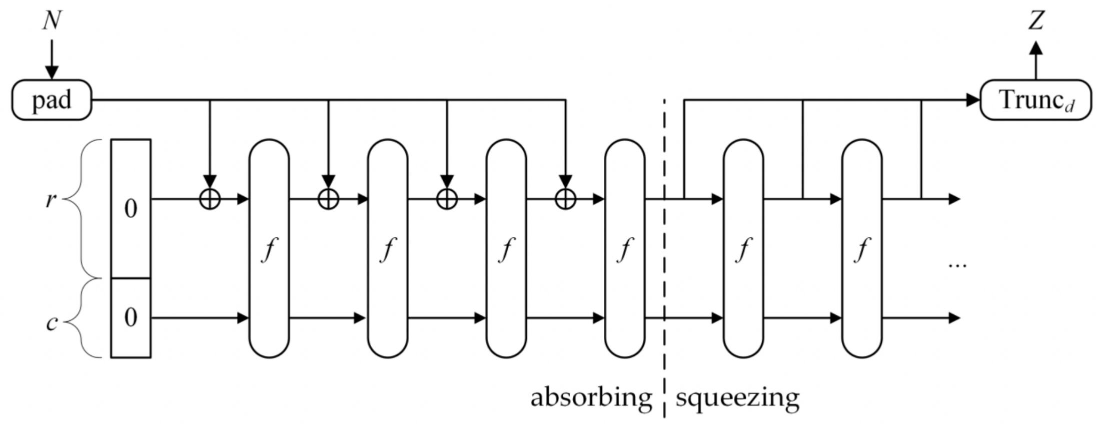


Figure 2.2. Model of sponge construction utilizing a KECCAK-$p[b,\ n_r]$ secure hash algorithm [1].

"pad10*1", with the asterisk symbolizing either an omitted zero bit or a repeated strings of zero bits. The number of zero bits needed to create a multiple of the rate can be calculated by (– len(N) – 2) modulo the rate. Calculating the padding rule is summarized by equation 2.16 through 2.18 [1].

The *rate* is a positive integer denoted by r, and defined as the total number of input bits called for encryption. In other words, it is the value of the length of bits partitioned from the message plus the padding rule, such that N+(pad(10*1) divided by the rate equals an integer. This step is taken to ensure that the message can be equally partitioned into strings of length r before entering the absorption phase. Likewise, the capacity is a positive integer and is denoted by c. The width b is greater than the rate r, and $b = r + c$. As previously stated, $N \parallel \text{pad}(r, \text{len}(N))$ is a multiple of the rate [1].

$$\text{Let } \ j = (-\text{len}(N) - 2) \bmod r \tag{2.16}$$

$$\text{pad}10^*1 = 1 \parallel 0^j \parallel 1 \tag{2.17}$$

$$\text{Let } \ N \parallel \text{pad}10^*1 = N \parallel \text{pad}(r,\ \text{len}(N)) \tag{2.18}$$

2.2.2. Absorption Phase. Before the padded message is XORed into the sponge function a string of zero bits is set equal to len(r) + len(c). During the absorption phase, a sub-string of the message, equal to the length of the rate, is XORed to the string of zero bits. The XORed string, concatenated to the string within the capacity, is converted into a state array and enters the underlying function KECCAK-$p[b, n_r]$. The next sub-string of the message, also equal in length to the rate, is XORed to the output of the previous KECCAK-$p[b, n_r]$ permutation. Once again, the XORed string is appended to the string representing the capacity and enters the underlying function KECCAK-$p[b, n_r]$. The capacity during the second iteration is determined by the previous output. The number of sub-strings that enter the sponge construction is equal to a multiple of the rate. This process continues until the entire message has in essence been "absorbed" into the sponge construction [1].

2.2.3. Squeezing Phase. The squeezing phase of the sponge construction gives the output of length d. After the last permutation of the absorbing phase, a string of length r is extracted from the output of the KECCAK-$p[b, n_r]$ permutation. If this string is greater than the length d, the excess bits are truncated. If the extracted string is less than length d, the remaining bits undergo another KECCAK-$p[b, n_r]$ permutation. A second string of length r is extracted and concatenated to the previous string of length r. Once again if the appended string is longer than length d, the excess bits are truncated. This process continues until the necessary length of the output d is satisfied. The entire function SPONGE[f, pad, r](N, d) can be expressed by the following steps [1].

$$\text{Let } \; P = N \,\|\, \text{pad}(r, \text{len}(N)) \tag{2.19}$$

$$\text{Let } \; n = \text{len}(P)/r \tag{2.20}$$

$$\text{Let } \; c = b - r \tag{2.21}$$

$$\text{Let } \; P_0, \; \dots , P_{n-1} \text{ be a string of length } r \text{ such that } P = P_0 \;\|\; \dots \;\|\; P_{n-1} \tag{2.22}$$

$$\text{Let } S = 0^b \tag{2.23}$$

$$\text{For } i \text{ from 0 to } n-1, \text{ let } S = f(S \oplus (P_i \parallel 0^c)) \tag{2.24}$$

$$\text{Let } Z \text{ be an empty string} \tag{2.25}$$

$$\text{Let } Z = Z \parallel \text{Trunc}_r(S) \tag{2.26}$$

$$\text{if } d \leq |Z|, \text{return Trunc}_d(Z); \text{else continue} \tag{2.27}$$

$$\text{Let } S = f(S), \text{and continue with Step (2.26)} \tag{2.28}$$

2.3. MERKLE TREES

Merkle Trees provide a way to organize data in such a way as to minimize the memory required by a CPU. In general, Merkle Trees can be visualized as inverted branches spawning from the first node known as a *root node*. From the root node, a series of further branching nodes extends such that all nodes can be traced back to the root node. The primary function of this structure is to allow stores of information, update information, and provide a reliable mechanism for queries to accurately retrieve information [2].

A series of messages, represented by $x_1, x_2, ..., x_n$, can be structured into a Merkle Tree. Each message under goes a SHA-3 function, after which the output from two neighboring messages can be concatenated and encrypted to form a new node. Once again, the outputs can be encrypted to form the next higher level node. The process continues until a root node is formed. Figure 2.3 displays the construction of a Merkle Tree, with the function $h()$ representing a SHA-3 encryption [2].

Subsequently, all the information contained in the lowest levels of the Tree is encrypted within the root. Any singular path can be chosen and it would contain all relevant information. This property allows for branches of the Tree to be trimmed to a single path, thus saving CPU memory. This technique is applied to the storage of all transaction history used in electronic currency to save memory while concurrently making the record of transactions immutable. Each block within the block chain represents one root containing all relevant information.

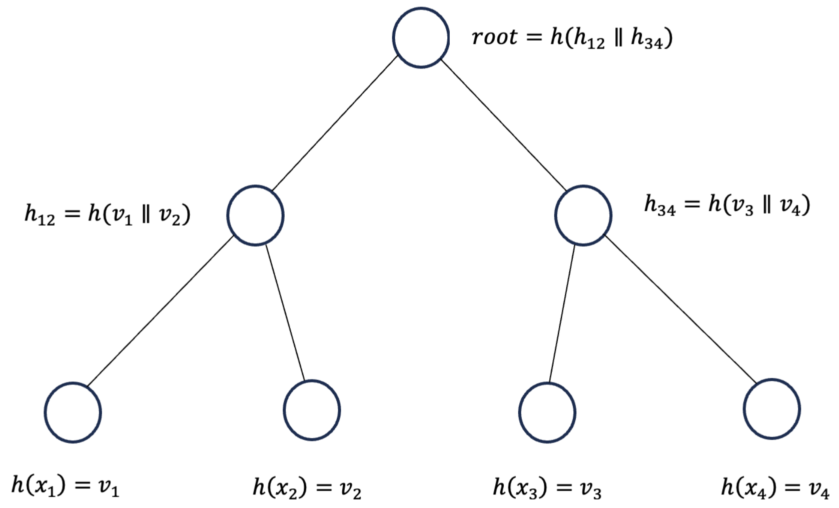

$root = h(h_{12} \parallel h_{34})$
$h_{12} = h(v_1 \parallel v_2)$
$h_{34} = h(v_3 \parallel v_4)$
$h(x_1) = v_1$
$h(x_2) = v_2$
$h(x_3) = v_3$
$h(x_4) = v_4$

Figure 2.3. An example of a Merkle Tree with higher level nodes encrypted from previously encrypted nodes [2].

3. CRYPTOGRAPHIC SIGNATURES

3.1. ELLIPTICAL CURVE DIGITAL SIGNATURE ALGORITHM

Crypto-currencies validate transactions through Elliptical Curve Digital Signature Algorithm (ECDSA). The algorithm begins with choosing a suitable curve. Often, implementation of ECDSA for electronic currencies implements the formula $y^2 = x^3 + ax + b \pmod{p}$, with the equation defined over the finite field $\mathbb{F}_p$. The field is defined with a prime number of elements to minimize the occurrence of zero as an output, hence the modulus operator. Both Bitcoin and Ethereum use the formula $y^2 = x^3 + 7 \pmod{p}$, however, random selection of an elliptical function can also suffice for increased security [7].

3.1.1. Point Addition. The algorithm makes use of point addition and point doubling. Point addition is described as the addition of two points that lie along the curve. Deriving the slope between the two points, for example point G and Q, yields a third point of intersection along the curve, denoted $-(G + Q)$. Reflection of point $-(G + Q)$ relative the x-axis yields a fourth intersection along the curve denoted $(G + Q)$. Figure 3.1 displays the the points G and Q along with their addition (G+Q) while the following equations 3.1 through 3.3 show steps for adding two points along an elliptical curve [8].

$$\text{slope} = \frac{y_G - y_Q}{x_G - x_Q} \tag{3.1}$$

$$x_{\text{(G+Q)}} = \{\text{slope}^2 - (x_G + x_Q)\} \pmod{p} \tag{3.2}$$

$$y_{\text{(G+Q)}} = \{\text{slope} \cdot (x_G - x_{(G+Q)}) - y_G\} \pmod{p} \tag{3.3}$$

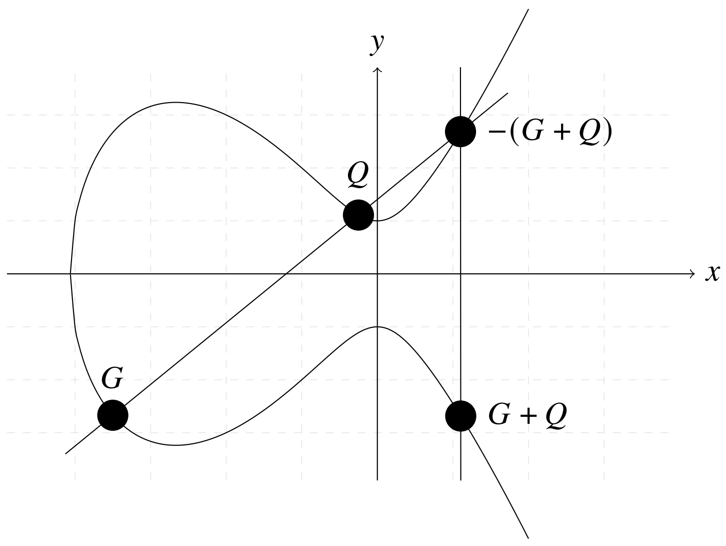


Figure 3.1. An elliptic curve over an infinite field displaying an example of point addition using the points G and Q.

3.1.2. Point Doubling. Point Doubling is simply the doubling of a single point that falls along the elliptic curve. A single point, for example point G, can be added to itself to yield the point 2G. The first derivative of the elliptical curve function at point G yields a tangential line that will intersect the curve at -*2*G. Once again, the reflection of point -*2*G relative to the *x*-axis will give the point *2*G. Figure 3.2 displays point G along with the multiple *2*G, while the following equations 3.4 through 3.6 show steps for point doubling along an elliptical curve. It should be noted that inverse function in the following equation 3.4 represents the modular multiplicative inverse. Furthermore, combining point doubling with point addition can significantly reduce computational time. If an integer, say *i*, is an odd number, point doubling can be used to compute the integer *i*-1. Point addition can subsequently be implemented to finish the computation (*i*-1)G+G = *i*G [8].

$$\text{slope} = \left\{\frac{d\,E(\mathbb{F}_p)}{dx} \cdot (2 \cdot y_G)^{-1}\right\} \pmod p \tag{3.4}$$

$$x_{(2G)} = \{\text{slope}^2 - (2 \cdot x_G)\} \pmod p \tag{3.5}$$

$$y_{(2G)} = \{\text{slope} \cdot (x_G - x_{(2G)}) - y_G\} \pmod p \tag{3.6}$$

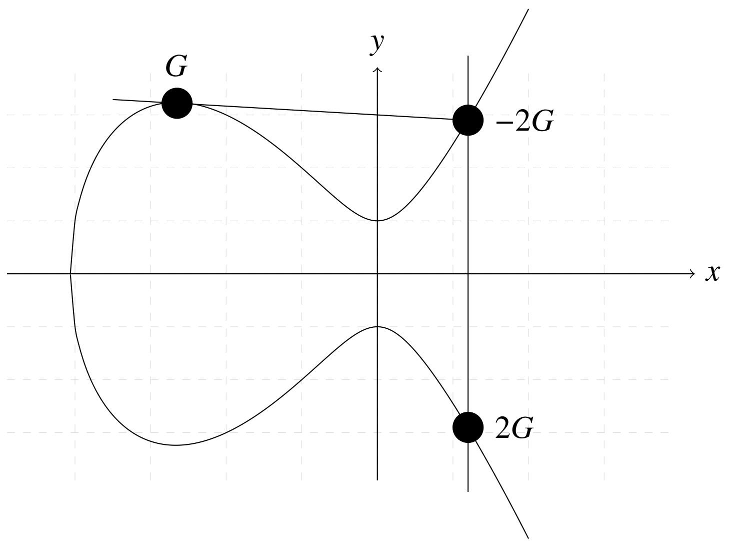


Figure 3.2. An elliptic curve over an infinite field with an example of point doubling using the base point G.

3.1.3. Domain Parameters. ECDSA consists of five parameters. First, the prime modulus, denoted p, defines the finite domain of the elliptic curve. the domain can be thought of as an x and y plane with p elements along the x and y axis. Any calculated value exceeding the value of p cycles back into the domain due the modulus operator. A point along the curve must be a subset of the elliptical curve defined in the finite domain, $G \in E(\mathbb{F}_p)$ [7, 8]

The next parameters is the prime order, denoted n. Given a randomly selected base point along the curve with the function existing within the finite field $\mathbb{F}_p$, n defines number of points possible to compute using point addition before arriving back at the initial base point. Subsequently, the cyclic addition of points along the curve arise with order n and subgroups of points with coordinates G to nG exist. Therefore, (n+1)G (mod n) cycles back to the beginning base point G. For all elliptical curve functions, the prime modulus minus one, (p-1) is a multiple of the prime order n. [7, 8]

The elliptic curve is also defined by the parameter coefficients a and b, observed in the equation $y^2 = x^3 + ax + b \pmod{p}$. Finally the last parameter is of course the base point, denoted as G and is comprised of the coordinates (x_G, y_G). Parameters of the ECDSA are shown in the following expression 3.7 [7, 8]. The following figure 3.3 displays the appearance of an elliptical curve when expressed over a finite field with p = 37 elements, and $E(\mathbb{F}_p) = y^2(\text{mod } 37) = x^3 + 7(\text{mod } 37)$, for x values 0 through 37.

$$E(\mathbb{F}_p) = (p,\ n,\ a,\ b,\ G) \tag{3.7}$$

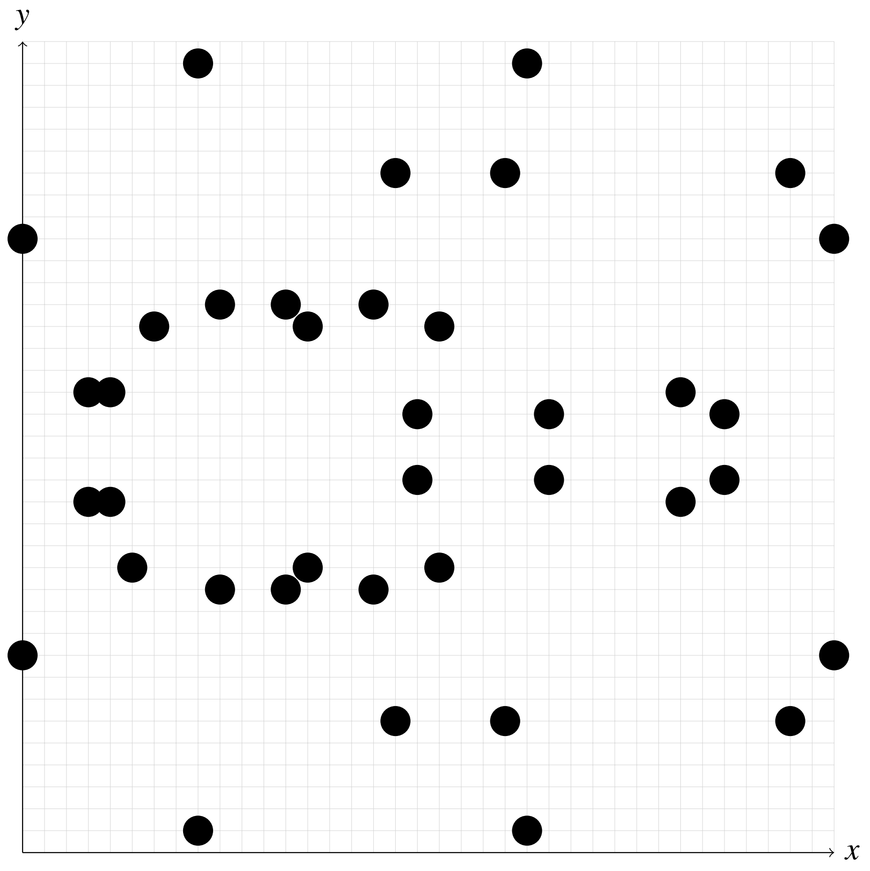


Figure 3.3. An elliptic curve output over a finite field.

3.2. VALIDATING A TRANSACTION

3.2.1. Key Generation and Signature. Generation of both public and private key begins with the selection of one random integer that becomes the private key, denoted k_{prv}, and falls between [1, n-1]. Using point doubling and addition, the integer k_{prv} operates on a randomly selected base point G. The output obtained from k_{prv}·G gives a second pair of coordinates Q that becomes the public key, denoted k_{pub}. After the generation of a public and private key, a transaction, otherwise known as a message, can be sent between participants [7].

An individual sending a transaction to another participant must provide a cryptographic signature to verify that the message was indeed from the intended sender. The signature begins with selecting a random integer *i* that falls between [1, n-1], Integer *i* is a separately selected integer that is not k_{prv} and must be changed for every transaction made. The following steps 3.8 through 3.11 provide the computations for the output variables (*r*, *s*), used to sign for a hashed message. The hashed message is converted to an integer equal to the length in bits, denoted as *t* [7].

$$iG = (x_i, y_i) : \text{ Then convert } x_i \text{ to integer } x_{int} \tag{3.8}$$

$$r = x_{int} (\text{mod } n) \tag{3.9}$$

$$s = \{i^{-1}(t + r \cdot k_{prv})\}(\text{mod } n) \tag{3.10}$$

$$\text{Sender signs transaction with output variables } (r, s) \tag{3.11}$$

3.2.2. Signature Verification. Once an individual has sent a message with a signature (r, s) to a recipient, the recipient needs to verify the sender's signature. First, the recipient receives a copy of senders domain parameters for the elliptic curve as well as the senders public key, recall $k_{pub} = k_{prv}$·G. The following equations 3.12 through 3.15 show the steps for a recipient of a message to validate the message came from the intended sender.

As one can see, the sender does not have to reveal their private key for signature validation as k_{prv} is not included within the equations [7, 8].

$$u_1 = (s^{-1} \cdot t)(\mathrm{mod}\, n) \tag{3.12}$$

$$u_2 = (s^{-1} \cdot r)(\mathrm{mod}\, n) \tag{3.13}$$

$$P = u_1 G + u_2 k_{pub} = (x_p, y_p) : \text{ Then convert } x_p \text{ to integer } x_{int} \tag{3.14}$$

$$v = x_{int}(\mathrm{mod}\, n) : \text{ Accept Signature if } v = r \tag{3.15}$$

4. BITCOIN

4.1. TRANSACTIONS THROUGH BITCOIN

Bitcoin was originally introduced as an electronic alternative to traditional methods of modern financial transactions, all of which depend on a trusted third party. The primary goal of Bitcoin is decentralization from financial institutions, accomplished through the ability to conduct and validate transactions within a trusted peer-to-peer network. In order to establish an honest network, the problem of double spending requires a solution to institute trust for every financial exchange. Double-spending is a fraudulent transaction in which a participant in the network reclaims the value of a transaction after receiving the product or service. Double-spending can only occur when a fraudulent block-chain out-competes the honest chain in length, with the longest chain accepted as the record of a valid transaction. In addition, records of past transactions must be agreed upon by participants within the network that cannot be subject to change. The following sections provide a review of Satoshi Nakamoto's proposal for an online peer to peer electronic cash system [9].

4.1.1. Financial Transactions in an Electronic Cash System. The first characteristic electronic cash systems require is the ability to allow monetary transactions. Transactions conducted in Bitcoin consist of the literal trade of a chain of digital signatures, which is considered to be the electronic "coin". When the owner of the coin signs the hash of the previous transaction and the public key of the payee, a coin is transferred to next owner with the signatures added to the end of the chain. However, the payee does not have the ability to confirm the coin has not been double-spent. In other words, the previous owner of the coin can spend the coin on multiple transactions, retracting the transaction before the payee is aware and receive the goods or services without paying [9].

In order to overcome the double spending problem participants within the system need to agree upon a record of all transactions that have occurred. If a reliable history is available to the network, any chain can be validated and confirmed that double-spending has not occurred. Therefore, providing a trustworthy history of all transactions is essential to the creation of the peer-to-peer network [9].

A timestamp sever offers a solution to double spending problem by providing a publicly published hash. The block of items within the hash are timestamped with the date and time of the transaction. The timestamp provides the necessary proof a transaction took place, and the publication provides the means for anyone to confirm that indeed the transaction did take place. Any further transactions will be added to the block-chain along with its own timestamp. For every timestamp that is added to the chain, the previous transactions comprising the block-chain are reinforced and obtain a greater level of security [9].

4.1.2. Validating Transactions through Proof of Work. In order to establish a working timestamp server, the proof-of-work within system provides a way to validate a transaction while simultaneously generating a record of the transaction. Once the required work is placed into validating a transaction, the date and time are placed into the hash. The proof-of-work begins with a predetermined value hashed and placed into the beginning of a block, called a nonce (number only used once). When this value is encrypted with a hash, a specific number of zero bits are returned to begin the block. The amount of work required to find the original nonce value increases exponentially with increasing numbers of zeros [9].

The CPU scans for a value that when found satisfies the number of zero bits. Once the required work is put into finding the nonce, the block cannot be changed, and the transaction is validated. As previously mentioned, the work required increases as the number of zero bits increases. If most of the network is comprised of honest CPU power the correct validated chain will grow at a faster rate, given that honest CPU power will begin building the chain earlier than competing fraudulent CPU power. The individual who mined for the nonce receives a financial incentive for supporting the transaction. Other computers within the network confirm the transaction by agreeing on the longest chain found that satisfies the work requirement. This cycle continues for each transaction, with a new nonce produced and added to the chain for each new transaction [9].

4.2. BITCOIN NETWORK

4.2.1. Participation Within the Network. The network is comprised of participating CPU's called nodes, which mine for a nonce by completing the proof-of-work. At the moment a transaction begins, it is broadcast across the entire network to all nodes. New transactions are placed into a block by each node. When the work requirements are satisfied by completing the proof-of-work, the node broadcasts to the rest of the network that it has found a solution. Other nodes accept the transaction if all previous transaction and the new transaction within block have not been double spent. Nodes then accept the block by beginning the process on the next transaction, creating a new block and subsequently producing a chain of recorded transactions. If two nodes find different solutions to the proof-of-work, the solution with the longest chain is accepted and added to the chain [9].

4.2.2. Creating Incentive to Participate Within the Network. A new coin begins with a unique first transaction that is validated and owned by the creator of the block. New coins that are mined create incentive for new nodes to join the network and participate in sustaining the currency. If the value of the coin begins to decrease a greater amount of work can be added to create a new coin, thus controlling for inflation. Incentive also occurs when validating a typical transaction between participants. The amount of work needed to validate a transaction is charged as a small fee, therefore sustaining participation with incentive to validate transactions. If the state of the network reaches a point where enough coins are in circulation, the incentive to participate can solely rely on validating transactions without incentive to create new coins [9].

The incentive for creating new coins and validating transactions protects against double spending. The amount of work required by a CPU to outpace the length of an honest chain and steal back a transaction is greater than the amount of work required to mine for coins and validate transactions. This systematic property protects against bad actors and encourages participants to stay honest [9].

4.2.3. Reclaiming Memory. As transactions continue to build within a block, the memory required maintain a chain also continues to increase. However, once a transaction is supported by previous validated transactions, a Merkel tree can be used to save space. To maintain the block's hash, a Merkel tree can be used to hash the previous transactions by trimming off the old transactions and keeping the block's root [9].

4.2.4. Further Discussion on Payment Verification and Privacy. Verification of transactions can be finalized without using a full network node. A full network node is a program ran on one's own computer that participates in the network by mining for coins and validating other transactions. A user without a full network node can search the network and find a record of the root and Merkel tree with the longest proof-of-work chain [9].

When combining multiple payments to reach a desired value, a single transaction can be carried out by summing multiple transaction until the value needed is reached. This summation of transactional values is referred to as splitting and combining. Multiple inputs can be placed into the network to sum the value needed and at most two outputs can be returned, one output for the value needed for the transaction and another output for the change if needed. Splitting and combining are analogous to traditional paper money being combined to a total sum and receiving the excess money as the change returned [9].

Privacy is still able to be maintained within the public network. While transactions are broadcast to the community, public keys used by individuals remain anonymous. Unless a participant displays their public key to the community, their identity cannot be known. However, for security reasons, it is recommended that an individual acquire a new public key for each transaction carried out. Given that a public key is randomly given for each transaction, acquiring a new public key would make the inference to a common owner impossible [9].

4.3. FRAUDULENT ATTACKS

Fraudulent attacks can only occur on a transaction initiated by the attacker. Value cannot simply be spontaneously created for an unspecified sum. The only fraudulent attack possible within the system is reclaiming a previously spent transaction, identified earlier as the double-spending problem. A fraudulent attacker would only be able to double-spend a coin if they were able to generate a longer chain in comparison to the length of an honest chain. The honest chain would begin the moment the transaction is broadcast to the network, by necessity, the attacker would have to "catch-up" to the honest chain [9].

The probability an attacker can reach the required length can be calculated as a Binomial Random Walk. The successful event of an honest chain extending by one block is represented by +1, while a failure event of the attacking chain extending by one is represented by -1. The following equation 4.1 expresses q_z, the probability that an attacker will catch up from a deficit. The variables p, q, and z represent the probability of honest node extending by one block, the attacking node extending by one block, and the number of blocks added after the transaction [9].

$$q_z = \begin{cases} 1 & p \leq q \\ (q/p)^z & p \geq q \end{cases} \tag{4.1}$$

The attacker's probability of catching up with the honest chain decreases exponentially as the number of blocks added to the honest chain increase. To ensure the payee securely receives the transaction, they need to wait until z number of block are added to the chain after the transaction. Each additional block increases the odds the transaction was added to an honest chain. Waiting for the appropriate number of blocks yields a Poisson distribution, with the average growth expressing the attacker's expected increase of blocks in equation 4.2 [9].

$$\lambda = z\left(\frac{q}{p}\right) \tag{4.2}$$

Multiplying the Poisson distribution by probability the attacker can catch up yields the probability the attacker can generate a longer chain and double-spend the coin. The variable *k* represents the current block position of the attacker's chain relative to the initial block the transaction was linked to. In other words, the expected progress the attacker has made is multiplied by the probability the attacker can match the length of the honest chain. The expression can then be rearranged to give the probability of a successful attack, displayed in equation 2.3 [9].

$$1 - \sum_{k=0}^{z}\left(\frac{\lambda^k e^{-\lambda}}{k!}\right)\left(1 - (q/p)^{(z-k)}\right) \tag{4.3}$$

5. ETHERUEM

5.1. EVM

Ethereum is a popular block-chain based currency used in today's market. Like Bitcoin, it is decentralized and offers the same protections against the double-spending problem. Ethereum validates its transaction through participant interaction with the Ethereum Virtual Machine (EVM), as opposed to participants within a network mining for transactions. While the EVM is stored on participating nodes within the network, the code to execute transactions are implemented through the EVM.

Ethereum is the first cryptocurrency that created a platform for the construction of smart contracts. Smart contracts are a unique feature of Ethereum that allows participants to construct executable code using a Turing-complete language, helping to facilitate transactions, or any other desired function between parties [6]. This property distinguishes Ethereum from other electronic currencies in that new applications are constantly being built, saved to the EVM, and available for participants to implement. The following subsections provide and overview of the Ethereum block-chain and how the EVM facilitates transactions between individuals.

Ethereum is a *state machine*, changing from one state to the next for every input executed. Transactions are stored as a Merkel tree with the tree path accepted as the canonical version, the true transactional history [6]. The *machine state*, distinct from the state machine, is capable of executing arbitrary code executed through the EVM [10]. The term state machine is generally used to describe the model of computation where the machine can only exist in one state at any given time. Alternatively, the *machine state* changes from block to block, carrying code the EVM can execute [10]. Ethereum is defined as distributed state machine, updating the state across the network through participants. The constantly updated and stored data is referred to as the World State, in other words, the current state of the entire network[6, 10]. The World State includes the history for all accounts, balances, and arrangements between participants. Only thorough the EVM can code be implemented by the machine state to update the World State [10].

Blocks within Ethereum function as a record of a series of transactions. A single block records a series of transactions which the EVM subsequently ”chains” the information to the previous block, thus the nomenclature ”block-chain”. A stack of chained blocks is known as a ”ledger”. Each transaction is collected into a block that undergoes the Ethereum state transition function, allowing arbitrary computation and the storage of the new state [6]. the World State can thus be viewed as being updated with the record of each transaction and relevant information. Therefore, Ethereum is also is a ledger of all transactions, blocks, and smart contracts that have been created [10]. The figure 5.1 gives a general overview of the machine state’s components which change for every transaction and who’s execution updates the World State.

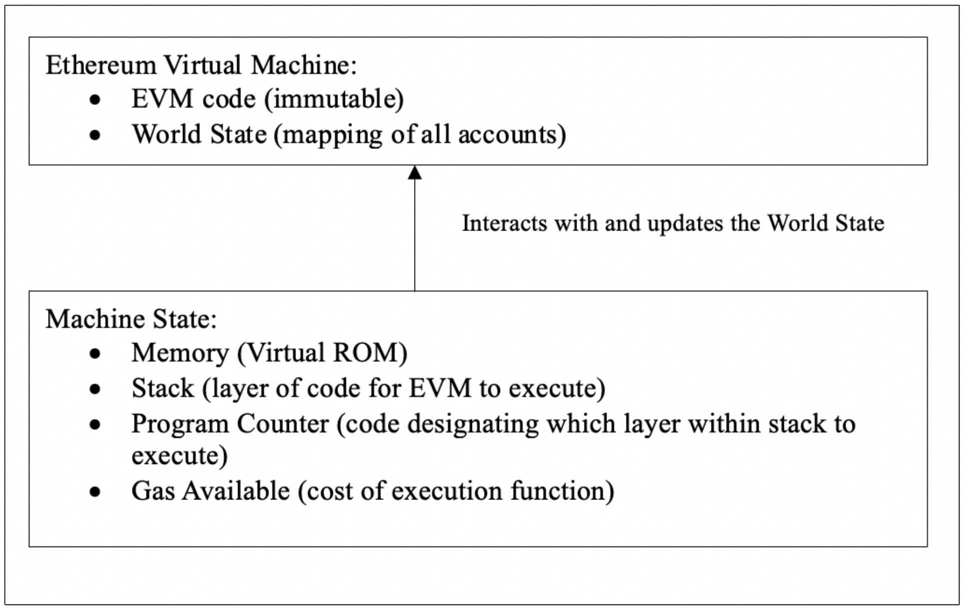

Ethereum Virtual Machine:
• EVM code (immutable)
• World State (mapping of all accounts)
Interacts with and updates the World State
Machine State:
• Memory (Virtual ROM)
• Stack (layer of code for EVM to execute)
• Program Counter (code designating which layer within stack to execute)
• Gas Available (cost of execution function)

Figure 5.1. Components of the machine state and its interaction with the EVM to update the World State

The machine state is capable of interacting with the EVM through the stack. The stack is comprised of items with each item defined as 256 bit word to a maximum depth of 1024 items [10]. The stack is used to hold code for the inputs and outputs of transactions and smart contracts. The program counter, located in the machine state, instructs which code should be executed by the EVM in a sequential order and is read from the memory [6]. The memory is destroyed once a call has been executed but initially receives program code from the virtual ROM [10]. Finally, the Gas is the transactional cost of code execution. Every transaction must pay a small fee for execution and the fee is referred to as the gas [6]. Once a transaction is executed, the World State is updated with the relevant information and the machine state becomes available for the next transaction. In this sense, the World State is transient with data distributed, stored, and updated amongst participating Ethereum nodes.

5.2. TRANSACTIONS

5.2.1. Overview of Transaction in Ethereum. Transaction are signed cryptographically and provide instructions to the Ethereum network to execute a particular function [6]. The most common instruction is simply sending the Ethereum currency, ETH, from one participants to another. Transactions are processed from two types of accounts; externally owned accounts (EOA) and contract accounts [4]. EOAs are accounts owned by participants in the network and include the account address, nonce, and a balance. Contract accounts are solely used to execute smart contract and do not belong to any participant. Instead, they are stored within the network and accessible to participants to utilize [4, 11]. EOAs are called upon for a transaction through a public key with owners having access to their accounts through a private key. Contract accounts differ from EOAs in that while stored on the block-chain, they contain code that can be executed when defined conditions are met [4, 11]. A more in depth discussion on smart contract will be discussed in a following section.

Table 5.1. Parameters associated with all EOAs and implemented during a transaction between participants [4, 5, 6]

.

Nonce	Transaction number of account
gasPrice	Wei paid per unit of gas expended for all computations executed
gasLimit	Maximum amount of gas allowed
To	160 bit address of the recipient of the transaction call
From	160 bit address of the sender of the transaction call
Value	Amount of Wei to be transferred to recipient
v,r,s	Values for cryptographic signature using ECDSA
input or data	Code for executing smart contracts or message call

Table 5.1 displayed above specifies the parameters associated with EOAs. Each field is required before a transaction can be sent or called upon by another account. The nonce is the total number of transactions the sender account as executed, which differs from the definition of nonce used in Bitcoin [6]. For a new transaction, this value would be the total number of transaction previously completed plus one. The gasPrice is determined by a unit called Wei, where Ether is define as 10^{18} Wei [6]. The gasPrice protects against infinitely recursive code by a attaching a cost proportional to the amount of computational power needed [12]. Thus, if a transaction exceeds the gasLimit, the transaction is cancelled with remaining monetary values returned to the sender's account. In addition to gasPrice and gasLimit, a 160 bit address of the recipient is included along with the amount of Ether to be transferred, computed in Wei [6]. Finally the recipient validates signature of the sender to confirm the transaction. The value r and s are used to validate the signature through the use of an Elliptical Curve Digital Signature Algorithm [6, 7].

The following figure 5.2 displays the flow chart when a transaction is called to be executed. The transaction can be thought of as being sent to the EVM, where by if a smart contract is called to be implemented, the argument will be executed on the transaction.

The program counter (PC) functions as a small memory for the order of instructions to be executed. The stack also contain ”slots” of memory for the input data where order is determined by the program counter [6]. Smart contracts are usually written in a higher language such as solidity, then compiled into operation code (opcode) [13]. The opcode removes or inserts items being held within the stack and subsequently deletes them after execution [6]. The PC and stack interact with the smart contract to order and briefly store the base opcodes needed to execute the function specified in the smart contract. After the operation is complete, the output of the transaction is stored within the block-chain.

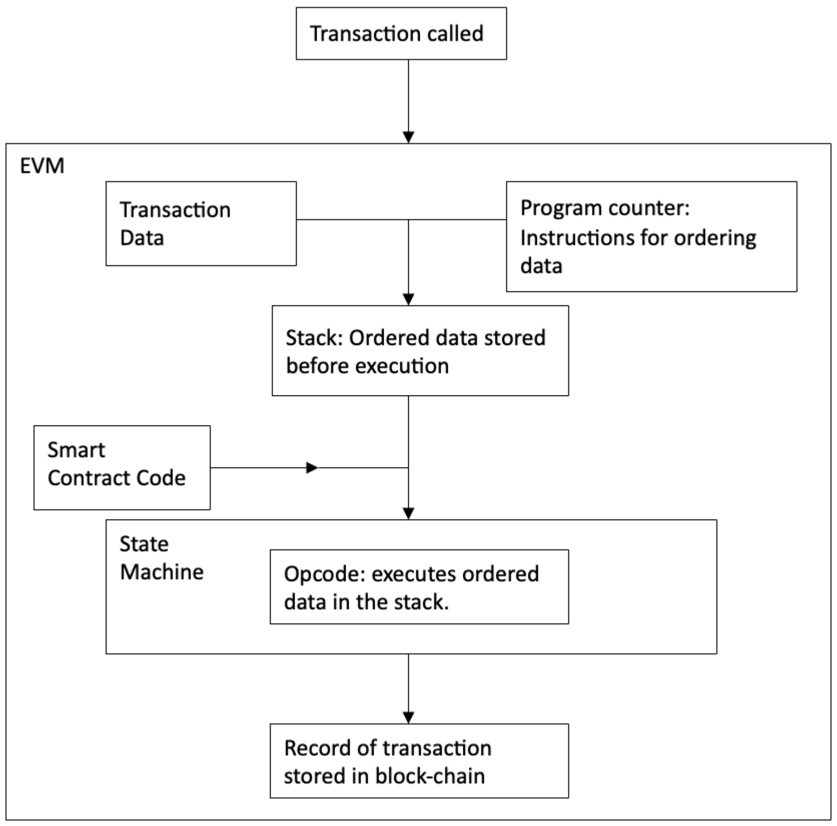

Transaction called
EVM
Transaction Data
Program counter: Instructions for ordering data
Stack: Ordered data stored before execution
Smart Contract Code
State Machine
Opcode: executes ordered data in the stack.
Record of transaction stored in block-chain

Figure 5.2. Flow chart of a transaction processed through the Ethereum Virtual Machine

5.2.2. Example of JSON Code for a Simple Transaction. The following code displays typical code used for a transaction between to participants. The first 17 lines shows the basic code when implementing a call from recipient to sender requesting a transfer of ETH. The function listed in line 2 as ”id” specifies two unique objects with in the argument; the account of the sender listed at ”from”, and the account of the recipient listed at ”to”. Line 3 specifies that JSON-RPC version 2.0 is being implement. The ”method”, specified in line four, states the the code is requesting a transfer of funds and also requires the signature, or private key, of the sender to finalize the transaction [14].

```
1    {
2    "id": 2,
3    "jsonrpc": "2.0",
4    "method": "account_signTransaction",
5    "params": [
6       {
7         "from": "0x1923f626bb8dc025849e00f99c25fe2b2f7fb0db",
8         "gas": "0x55555",
9         "maxFeePerGas": "0x1234",
10        "maxPriorityFeePerGas": "0x1234",
11        "input": "0xabcd",
12        "nonce": "0x0",
13        "to": "0x07a565b7ed7d7a678680a4c162885bedbb695fe0",
14        "value": "0x1234"
15      }
16     ]
17    }
```

The code implemented for the sender to sign a transaction is given in the following code below. Within line 5, the ”raw” argument gives the signed transaction in Recursive Length Prefix (RLP) . The RLP is how the machine state updates the world state by storing the data in the block [14]. The data is then saved using a Patricia Merkle Trie [6]. The parameters for the transaction ”tx” include signature of the sender, specified the the values v, r, s. While r and s were discussed in ECDSA, the parameter v defines the point of origin as either odd of even, given that the symmetry of an elliptical curve allows the x coordinate to have two variables along the y-axis. Finally, the transaction with the hashed signature is specified in line 17.

```
1   {
2   "jsonrpc": "2.0",
3   "id": 2,
4   "result": {
5   "raw":
"0xf88380018203339407a565b7ed7d7a678680a4c162885bedbb695fe080a44401a
6e400000000000000000000000000000000000000000000000000000000000000122
6a0223a7c9bcf5531c99be5ea7082183816eb20cfe0bbc322e97cc5c7f71ab8b20ea
02aadee6b34b45bb15bc42d9c09de4a6754e7000908da72d48cc7704971491663",
6   "tx": {
7   "nonce": "0x0",
8   "maxFeePerGas": "0x1234",
9   "maxPriorityFeePerGas": "0x1234",
10  "gas": "0x55555",
11  "to": "0x07a565b7ed7d7a678680a4c162885bedbb695fe0",
12  "value": "0x1234",
13  "input": "0xabcd",
14  "v": "0x26",
"r":
"0x223a7c9bcf5531c99be5ea7082183816eb20cfe0bbc322e97cc5c7f71ab8b20e",
"s":
"0x2aadee6b34b45bb15bc42d9c09de4a6754e7000908da72d48cc770497
1491663",
"hash":
"0xeba2df809e7a612a0a0d444ccfa5c839624bdc00dd29e3340d46df
3870f8a30e",
      }
    }
  }
```

Following the signature and the transaction becoming cryptographically hashed, the transaction is broadcast to the network. Once broadcast, the transaction is held within a pool of other transactions awaiting validation. A participant within the network, known as a validator, chooses which transactions to validate based upon greatest amount of gas they can receive for successfully completing the transaction. In other words, the validator is paid a small amount of monetary value for completing the transaction. With the transaction completed, the history of the transaction is stored within the block-chain. The greater the number of block added to the chain, the greater the security the transaction cannot be altered.

5.3. SMART CONTRACTS

5.3.1. Overview of Smart Contracts. Smart contracts are implemented in transactions between either an EOA and a contract account, or between two contract accounts. A contact account is a special type of account that holds coded instructions to execute a particular algorithm desired by a participant or participants [11, 15]. For example, if two individuals make a bet, a smart contract can be created to execute a transaction at the moment the event has been completed. This allows for an unambiguous execution of the code and resolves any discrepancy between participants. However, in practical application, a smart contract can be coded to execute any arbitrary computation.

An important feature of smart contracts is the ability to call other smart contracts, increasing the versatility a single computation cannot achieve on its own. Each smart contract is assigned its own address within the block-chain and accessible to the EVM. Once incorporated into the block-chain the code becomes immutable [10]. If a participant wants to call a previously created smart contract, a message can be sent to the address to implement the contract in a transaction. While several programming languages are used to create smart contracts the most popular is Solidity, created solely for Ethereum and the creation of smart contracts [13].

5.3.2. Uniswap Smart Contracts. Uniswap is trading platform on Ethereum that allows participants to trade various tokens. Tokens are typically defined as any digitally owned asset. For example, Bitcoin, ETH, and NFT's, are all examples of tokens as well as video game winnings. If a value can be coded it can be represented as a token. Many trades are made with fungible tokens standardized by ERC-20 tokens. The purpose of ERC-20 tokens is to allow values of fungible tokens to be interoperable with any other type of fungible token. The ERC-20 utilizes an Application Programming Interface (API) to facilitate the exchange of tokens. Tokens that are able to be traded and exchanged on Uniswap are categorized as ERC-20 tokens [16].

Uniswap is comprised of two components, periphery and core contracts. Core contracts contracts hold reserve assets and therefore need to be secure and separate from the additional functions stored within periphery contracts. The periphery contracts contain code needed to swap one token for another. Liquidity providers purchase a set amount of two types of tokens that are capable for being traded, such as token A and token B. In return, the liquidity providers receive a third token referred to as a *liquidity token*, signifying partial ownership of the trading pool. The available trading pool fluctuates over time leading to a decrease or increase of token value. Traders then utilize the pool of tokens to exchange one token for another with the goal of achieving the best exchange rate. A small fee is provided to the owners of the liquidity tokens for each exchange [17, 18].

An exchange begins with a periphery account called and transferred a specified amount of a token. The periphery account will be a smart contract capable of executing a swap function on the tokens desired to be exchanged. The smart contract will identify the amount to be exchanged and determine the initial destination for the output, token B. The initial output is the address of the first token to be swapped, token B. Many times, if multiple exchanges are required with multiple tokens, known as the path, there will be an address for each output token needed. Along the path, the input token is sent with the exchange swap function called to trade the token. The final destination of an exchange will be the address of the trader [17].

The core contract determines the number of out tokens available plus the reserves. The reserves are determined by the total number of output tokens available minus the number of input tokens called for exchange. The core contract sends the final amount to the trader's address and updates the available output tokens with the value of reserves [17].

5.3.3. Example of Uniswap Core Contract. The following examples specifies the arguments for a trade between two tokens and displays the base arguments used in the core contracts . The first step is to call the necessary interfaces and libraries. In this template example, the UniswapV2Pair.sol core contract is implemented to first gain access to the liquidity pools [17].

```
pragma solidity =0.6.12;

import './interfaces/IUniswapV2Pair.sol';
import './UniswapV2ERC20.sol';
import './libraries/Math.sol';
import './libraries/UQ112x112.sol';
```

```
7   import './interfaces/IERC20.sol';
8   import './interfaces/IUniswapV2Factory.sol';
9   import './interfaces/IUniswapV2Callee.sol';
```

The contract UniswapV2Pair.sol is a core contract that allows for the exchange of tokens. It calculates the amount needed for an exchange along a path. Given over 3.8x10^5 ERC-20 tokens exist, not all tokens are paired with one another. Instead, a path of multiple exchanges can be made to reach the desired output. UniswapV2Pair.sol also allows the minting of liquidity tokens, the creation of new token pairs. After the creation of liquidity token the core contract updates the amount in reserves. The UniswapV2Pair.sol is the central contract that allows access to the liquidity pools. Periphery contracts then will call the core contracts to add liquidity token, or receive output tokens for an exchange [17].

The UniswapV2ERC20.sol is another core contract implemented that maintains similar functionality as UniswapV2Pair.sol, but is solely focused on creating liquidity pairs for ERC-20 tokens. This core contract maintain the pools of liquidity tokens for ERC-20 pairs and their functions. The library Math.sol simply contains functions that allow for the common mathematical functions needed to compute exchanges. Within line 7, the IERC20.sol contains the standard criteria necessary for a token to be categorized and function as an ERC-20 token. Another core contract listed is the UniswapV2Factory.sol, which creates an address or registry for all token pairs. Finally, the UniswapV2Callee.sol allows users to borrow tokens from one pool temporarily, whats known as a flash swap, then return the tokens within a single transaction. The next lines of code specify the variables required before adding addition functionality to a core contract [17].

```
11  contract UniswapV2Pair is IUniswapV2Pair, UniswapV2ERC20 {
12      using SafeMath  for uint;
13      using UQ112x112 for uint224;
14      uint public constant MINIMUM_LIQUIDITY = 10**3;
15      bytes4 private constant SELECTOR =
           bytes4(keccak256(bytes('transfer(address,uint256)')));
16
17      address public factory;
18      address public token0;
19      address public token1;
20
21      uint112 private reserve0;
22      uint112 private reserve1;
23
24      uint32  private blockTimestampLast
25
26      uint public price0CumulativeLast;
27      uint public price1CumulativeLast;
28
29      uint public kLast;
30      }
```

UniswapV2Pair is equivalent to IUniswapV2Pair and works in conjunction with UniswapV2ERC20. SafeMath is a library that protect against overflows and underflows, both of which result in values that fall outside the range possible data values. Fractional computations are not supported by the EVM. Therefore, to overcome this limitation, Uniswap uses the UQ112x112 for uint224, listed in line 13. This function is 224 bits split into two 112 bit values. The first value of 112 bits represent an integer while the second 112 bits represent the following decimal value, such as 1.5. The minimum number of liquidity tokens is held constant at 1,000 as defined in line 14. Line 15 gives the ABI selector that allows the core contract to interact with external contracts and periphery contracts [17].

Line 17 gives the address to the factory contract. The factory is the point where a call can access the pool of two available tokens. Lines 18 and 19 give the address of the two tokens, with token0 expected to be exchanged for token1. Lines 21 and 22 give the reserves, or amount of available tokens within the pool capable of being exchanged. The following argument in line 24 gives the timestamp of the last exchange to have occurred [17].

The public price of token1 and token2, displayed in line 26 and 27, is used to calculate the average exchange rate of the tokens relative to one another. The exchange rate is calculated using the constant *kLast*, which is defined as *reserve0*reserve1*. This value remains constant for each exchange, therefore a decrease in the reserves of a token necessitates an increase in value for the same token. The objective of the trader is to utilize the liquidity pool with the best exchange rate. Some participants generate profit by finding the pool with the lowest rate, then utilizing another pool to trade a token at a higher rate. The following figure 5.3 displays the change of two token quantities relative to one another in relation to the constant *kLast* [3].

As shown in the figure 5.3, as quantity of reserves for token A increase, the ratio $kLast/x$ decreases. Thus, the value of token B will have increase to achieve a fair exchange rate [3]. The example code given above outlines the requirements for Uniswap core contract. A variety of functions can be added, such as addition or removal of liquidity, or the creation of token pairs.

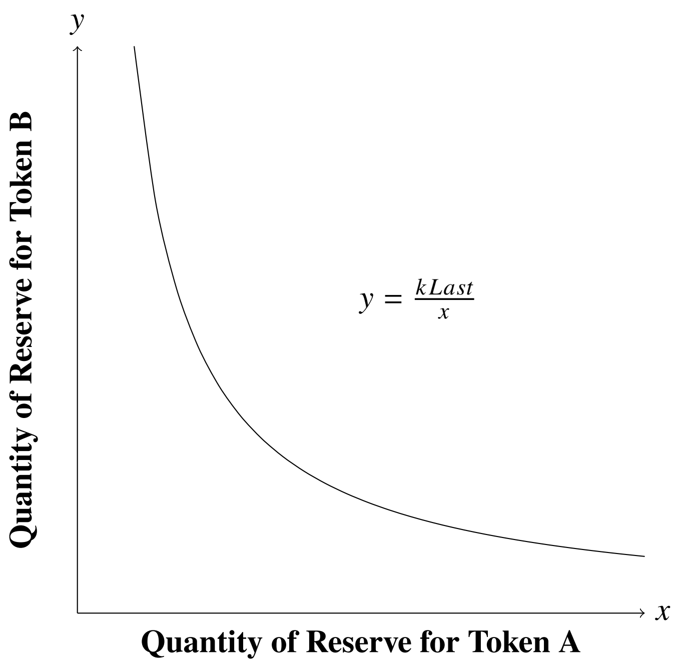


Figure 5.3. An example of function utilizing *kLast* to calculate the reserves of either token A or token B [3].

5.3.4. Example of Periphery Contract. The next lines of code are an example of an exchange that is initiated through the periphery contract UniswapV2Router02.sol [17].

```
pragma solidity =0.6.12;

import '@uniswap/v2-core/contracts/interfaces/IUniswapV2Factory.sol';
import '@uniswap/lib/contracts/libraries/TransferHelper.sol';

import './interfaces/IUniswapV2Router02.sol';
import './libraries/UniswapV2Library.sol';
import './libraries/SafeMath.sol';
import './interfaces/IERC20.sol';
import './interfaces/IWETH.sol';
```

In line 4 the library Transfer Helper facilitates the interaction with ERC-20 tokens and negates the return of true or false outputs when returning output tokens. The IUniswapV2Router02 is the main periphery smart contract. It computes the amount of tokens necessary for each trade along a path. It is also capable of creating a new pair to exchange as well as call core contracts to access tokens. The other periphery contract implemented is UniswapV2Library.sol. This contract allows a user to calculate the amount of output token they will receive for a trade. It can also be implemented to find the best path of exchanges to maximize the return of an output token [17]

ETH is not considered an ERC-20 token and line 9 specifies the IERC20 function that ensures enough tokens are available for an exchange. Because ETH is not considered an ERC-20 token, IWETH.sol exchanges ETH for what is known as wrapped ETH or WETH, capable of interacting with other ERC-20 tokens. These functions are the core of the periphery contract and allow the implementation of ERC-20 token exchange. The next lines of code executes required variables before functions can be executed when exchanges ETH [17].

```
11  contract UniswapV2Router02 is IUniswapV2Router02 {
12      using SafeMathUniswap for uint;
13
14      address public immutable override factory;
15      address public immutable override WETH;
16
17      modifier ensure(uint deadline) {
18          require(deadline >= block.timestamp, 'UniswapV2Router:
            EXPIRED');
19          _;
20      }
21
```

```
22     constructor(address _factory, address _WETH) public {
23        factory = _factory;
24        WETH = _WETH;
25     }
26
27     receive() external payable {
28        assert(msg.sender == WETH); // only accept ETH via fallback
29        from the WETH contract
30     }
```

Lines 14 and 15 state the address of the factory and WETH are immutable. The address are then defined at lines 22 through 24. Setting them as immutable ensures they cannot be tampered with once the contract is saved to the block chain. The modifier in line 17 ensures the contract is executed within the appropriate time frame. Last, the receive external payable in line 27 is used to receive the output token ETH, after being redeemed from WETH. The final lines of code presented for peripheral contracts gives a simple exchange between two tokens [17].

```
32  function swapTokensForExactTokens(
33     uint amountOut,
34     uint amountInMax,
35     address[] calldata path,
36     address to,
37     uint deadline
38  ) external virtual override ensure(deadline) returns (uint[] memory
39     amounts) {
40     amounts = UniswapV2Library.getAmountsIn(factory, amountOut, path);
41     require(amounts[0] <= amountInMax, 'UniswapV2Router:
42        EXCESSIVE_INPUT_AMOUNT');
43     TransferHelper.safeTransferFrom(
44        path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0],
45        path[1]), amounts[0]
46     );
47     _swap(amounts, path, to);
48  }
```

This function, beginning at line 32, swaps a token for a specified amount of an output token. A user would specify the amountOut desired and the maximum amount of token they are wiling to spend for an exchange fee. A call data path in line 35 is needed as multiple exchanges might be required to reach the desired output token. Line 36 defines the final address the output token are to be received. Next, line 40 defines the amount of token to be purchased for each step along the path. If the maximum amount of output token is not enough to meet the desired output, the transaction is reverted. The final function for TransferHelper, starting in line 43, allows the call swap to transfer the initial ERC-20 token to the first pair exchange account [17].

6. PARTIAL LEAST SQUARES REGRESSION

6.1. INTRODUCTION TO PLSR

Partial Least Squares regression (PLSR) is a statistical technique that can be utilized when high levels of correlation or dependence exists between independent variables. PLS is categorized as a type of supervised linear dimension reduction that uses response variable$y$ and the matrix of predictors $\mathbf{X}$ to generate linear combinations of weighted predictive variables, called latent variables [19]. While Principal Component regression is similar to PLS and uses the term loading to express a predictor's contribution to a component, PLSR does not generate orthogonal loadings. As such, a predictor's effect on a component is referred to as a weight in PLSR [20].

When computing latent variables from predictors, denoted as z_m, PLSR uses decomposition to generate linear equation and maximize the covariance of predictors in relation to the response variable y [20]. In other words, this method finds new directions of vectors that maximize the predictor's relation to y in order to explain the variance in both y and $\mathbf{X}$ [19, 20]. Each latent variable generated is a linear combination of the predictor variables. The latent variable, or multiple latent variables, are then used to generate a component. Finally, all predictor variables within the component are set orthogonal with respect to the latent variable to ensure the component explains the covariance of y and $\mathbf{X}$ to the greatest degree [20]. The result is a sequence of latent variables, z_1 through z_m, that remain orthogonal to one another when generating multiple components [19]. The following sections will provide a more detailed explanation of PLSR and why they can be used for highly correlated or dependent predictor variables.

6.2. COMPUTING THE COMPONENTS

PLSR first begins with each predictor variable standardized to a mean of zero and standard deviation of one. After the standardization of the input variables, the contribution of each predictor variable, denoted x_j, with $j = 1, ..., p$, is determined for the component. Equations 6.1 through 6.5 give the steps to compute the first component [19].

$$\hat{\varphi}_{1j} = \langle x_j, y \rangle \tag{6.1}$$

$$z_1 = \sum_{j=1}^{p} \hat{\varphi}_{1j} x_j \tag{6.2}$$

The previous two equations give steps for computing the weights associated with the first component. Within equation 6.1, $\hat{\varphi}_{1j}$ is computed from the inner product of x_j in relation to the response variable y and quantifies the direction a predictor has relative to the y. This step provides the weights that are applied to a predictor variable, and shrinks the contribution of a predictor if its correlation, or direction, is lower relative to other predictors when summed to find z_1. The first latent variable z_1 can thus be viewed as linear combination of each predictor variable x_j multiplied by its weight, $\hat{\varphi}_{1j}$. The summation of these variables yield the maximum covariance and direction of the first PLSR. The next step is to regress y onto z_1 followed by computing the first prediction $\hat{y}^{(1)}$ [19].

$$\hat{\theta}_1 = \frac{\langle z_1, y \rangle}{\langle z_1, z_1 \rangle} \tag{6.3}$$

$$\hat{y}^{(1)} = \bar{y}\mathbf{1} + \hat{\theta}_1 z_1 \tag{6.4}$$

Within equation 6.3, the inner product $\langle z_1, y \rangle$, specifies how much of the variance in the output y is explained by z_1. Dividing this value by the squared norm, $\langle z_1, z_1 \rangle$, normalizes the variance and scales the regression. Equation 6.4 then finds the fitted regression line with the coefficient $\hat{\theta}_1$ multiplied by the latent variable z_1. It should be noted that $\bar{y}\mathbf{1}$ represents the average y multiplied by vector of ones. If we were to find the second component,

$\hat{y}^{(1)}$ would be used to compute $\hat{y}^{(2)}$ with $\hat{y}^{(2)} = \hat{y}^{(1)} + \hat{\theta}_2 z_2$. Finally, the last step is to orthogonalize each predictor x_1 through x_p with respect to the latent variable z_1. This final step, shown in equation 6.5, ensures the original predictors will now have no correlation to the latent variable [19].

$$x_j^1 = x_j - \frac{\langle z_1, x_j \rangle}{\langle z_1, z_1 \rangle} z_1 \tag{6.5}$$

As shown in the previous equations, initial predictor variables are used to construct a new latent variable that is weighted. Each latent variable is then optimized to a maximum correlation to the output variable y. Equations 6.1 through 6.5 can be expanded to included components beyond the first component, each of which would be orthogonal to one another. If by chance, each input variable was orthogonal to each other, PLSR would simply yield the least squares estimate. Steps to find second component any beyond are given in the following equations. Each equation assumes that all predictor variables have been standardized. Also, it is defined that $\hat{y}^{(0)} = \bar{y}\mathbf{1}$ and $x_j^{(0)} = x_j$. Each predictor and component is represented by $j = 1, ..., p$ and $m = 1, ..., p$ respectively. After completing the steps, sequencing the fitted vectors $\hat{y}^{(m)}$ for all necessary components will give the best prediction [19].

$$z_m = \sum_{j=1}^{p} \hat{\varphi}_{mj} x_j^{m-1} \tag{6.6}$$

$$\hat{\theta}_m = \frac{\langle z_m, y \rangle}{\langle z_m, z_m \rangle} \tag{6.7}$$

$$\hat{y}^{(m)} = \hat{y}^{(m-1)} + \hat{\theta}_m z_m \tag{6.8}$$

$$x_j^m = x_j^{m-1} - \frac{\langle z_m, x_j^{m-1} \rangle}{\langle z_m, z_m \rangle} z_m \tag{6.9}$$

6.3. INTERPRETING THE OUTPUT

From the previous equations, it can be seen in equation 6.8 that the fitted value for m components would equate to $\bar{y}\mathbf{1}$ plus the summation of $\hat{\theta}_m z_m$ from $m = 1, ..., p$. This result is indeed a linear equation with $\hat{\theta}$ representing the coefficient and z_m representing the latent variable. Often when the output is generated after computing a PLSR, the coefficient associated with the latent variables will be retrieved. While this can be useful to understand the contribution of each latent variable, it does not provide an explanation for the contribution of the original predictors' effects on each component [19, 20].

A more useful metric often deployed to understand components is the accumulative percent variance explained with the incorporation of each subsequent component. As more components are implemented within a model, the percent variance of the response variable y explained by component's incorporation will increase. Therefore, this metric is also used to determine the number of components needed within the model. At a certain point, the number of components implemented will be exhausted and further components incorporated will not yield an increase in variance explained [19, 20].

To understand the effects of each predictor, the weights used to compute a latent vector are represented by the variable $\hat{\varphi}_{mj}$, displayed in equation 6.6. As previously mentioned, computing a latent variable z_m is executed by the summing the product of the weight and predictor variable. The absolute value of the weight can then be used to quantify the contribution of a predictor variable within a component. The sign associated with a predictor, either positive or negative, represents the directionality of the weighted predictor's effect on the component [19, 20]. These three outputs are obtained during the PLSR on bitcoin value using on-chain data.

7. MODELING OF BITCOIN VALUE DURING 2020 HALVING EVENT USING PLS REGRESSION

7.1. INTRODUCTION TO BITCOIN HALVING

During the creation of Bitcoin, Satoshi Nakamoto limited the maximum number of Bitcoins to a value of 21 million [9]. In order to delay circulating Bitcoins from reaching the maximum value, approximately every four years the reward for successfully mining a transaction or block is reduce by one half. This period of halving repeats itself for every 210,000 mined Bitcoin blocks. For example, in 2012 the emission of Bitcoin for every new block found was reduced from 50 to 25 Bitcoin. Likewise, in 2016 the reward for finding a new block was reduced from 25 to 12.5 Bitcoin [21]. The effect of limiting the supply of Bitcoins to be mined is often anticipated to increase the value of the cryptocurrency. The next halving event is expect to occur mid April 2024, incentivising individuals to invest before the halving event in order to capture the rising value from the subsequent bull market. However, even though the FTC has approved Bitcoin as an exchange-trade fund, such factors such as inflation and increasing interest rates can effect investors' trust in placing their assets within Bitcoin.

We have attempted to model the Bitcoin price (USD) of the May 11th 2020 Bitcoin halving using Partial Least Squares (PLS) Regression to provide insight on the changing value of Bitcoin during high volatility. The time period we chose to analyze was a one year period, November 11th 2019 through November 9th 2020, with the halving event directly between each date. The variables from the on-chain data used for the analysis were the reward in USD, the fee in US dollar, difficulty, and the number of transactions within a block. The following sections describe the methods used for obtaining and analyzing the data, as well as a prediction model for a 20-day moving average of Bitcoin value using PLS regression.

7.2. OBTAINING ON-CHAIN DATA AND DATA VISUAL ANALYSIS

7.2.1. Obtaining On-Chain Data. To acquire on-chain data of Bitcoin, most individuals utilize an extract, transform, and load (ETL) to gain access to the data. However, this method requires a user to become an active participant and download the entire block-chain record of verified transactions, requiring substantial memory. Only then can one acquire a username and password to access the local host for on-chain Bitcoin data. To circumvent this requirement, the site blockchair.com/dumps uploads daily records of all verified transactions within a block. A script in python was written to download all the data files associated with each date within the defined length of time. The script also contained code to append the data for each date to one another, generating a complete dataset for each block represented by a total of 52,997 rows.

To obtain the closing value of Bitcoin for each day, a file was downloaded from Yahoo finance where the necessary dates were selected. Before these two dataset were combined, a general visual analysis of Bitcoin value in USD were generated with a moving average of twenty days.

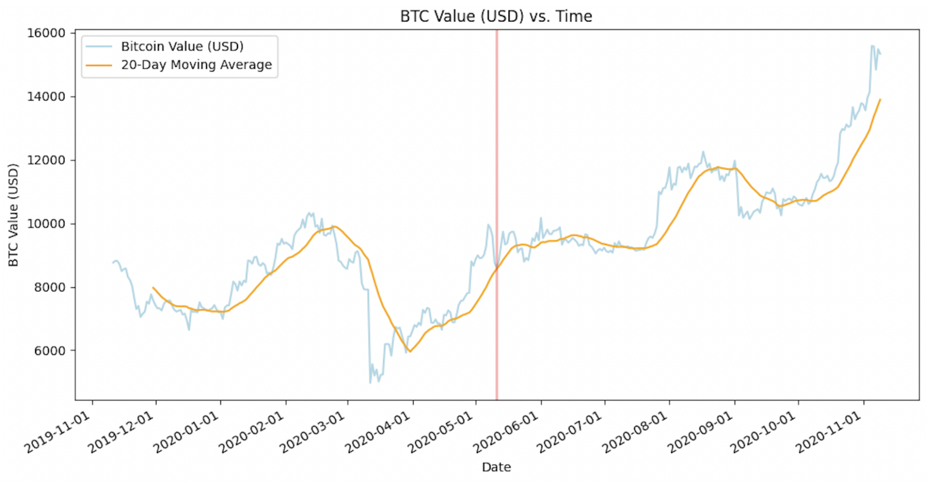


Figure 7.1. The value and moving average of Bitcoin in USD every day for six months prior and after the halving even in May.

A twenty day moving average was placed over the raw data to highlight trends. Form the generated plot displayed in figure 6.1, the value of Bitcoin significantly drops two months prior to the halving event, represented by the red vertical line. Following the halving event, the value of Bitcoin rises then enters a period of exponential growth in the final months. We decided to use the twenty day moving average of Bitcoin value as the variable of interest, using on-chain data to predict the moving average value. Through out the study all form of analysis were completed using scripts written in Python.

7.2.2. Predictor Variables and Data Visualization. Four variables from or derived from on-chain data were selected for study; rewards in USD, difficulty, fee in USD, and transaction count. Rewards, difficulty and transaction count are all found within on-chain data. Fees are calculated from on-chain data. While rewards and fees are initially recorded in the smallest unit of value, Satoshi, the data can be converted to USD. The dataset used in this study already converted this value and was utilized for the sake of convenience. For each of the four variable obtained, three new variables were generated; summation of the variable per day, average value of the variable per block, and the average value of the variable per transaction. The following table 6.1 provides definitions for each of the on-chain variables.

Table 7.1. Definitions for each variable obtained that originated from on-chain Bitcoin data

Reward	Newly generated Bitcoin plus the transaction fee
Difficulty	Computational power needed to verify or mine a transaction
Transaction Fee	Input Bitcoin value minus the output Bitcoin value
Transactions Count	Number of transactions mined on one block

Summing each variable was accomplished by grouping each variable by date and calculating the summation upon each variable. The averaged value per block was calculated by summing the variable, grouped by date, and dividing this value by total number of blocks verified within the same date. Finally, the averaged value per transaction, was calculated by grouping the data by date and summing the variable as well as summing the number of transaction. The summed variable was then divided by the summation of the transaction count. Given four variable from the block-chain and three calculations for each variable, twelve variables were obtained. However, dividing the averaged number of transactions per transaction equates to one, therefore it was removed as a variable. A 20-day moving average for each newly generated variable was calculated and appended to a new data set, along with the moving average of Bitcoin. The moving average of each variable was intended to be utilized as predictor variables in the PLS regression. In total, eleven moving average variables to predict the moving average of Bitcoin value were generated. The following subsections discuss each predictor variable.

7.2.3. Rewards (USD). First, data visualization based on rewards were evaluated and are displayed in figure 6.2. Viewing the total rewards all miners received in one day, values fall between the approximate range of 8 to 20 million USD. At the point of halving, represented by the vertical red line, the rewards can be seen to significantly drop. During periods of high trading volume, a greater number of blocks will be verified per day given a limit to the amount of available memory per block. When observing the moving average of rewards per transactions, maxima and minima are less accentuated as expected. While forty or more USD seems high for validating a transaction, rewards are also given for verifying a block and thus add to the overall value of rewards received.

When compared the value of Bitcoin, rewards can be seen to closely mirror its trend. Rewards can be observed to reach a local maxima around February, May, and August. However, the halving event forces available rewards for mining a Bitcoin to briefly move in the opposite direction before regaining a similar movement to Bitcoin. This sudden opposing trend, yet otherwise correlating trend, negates the rewards' capacity to be a strong predictor of Bitcoin value.

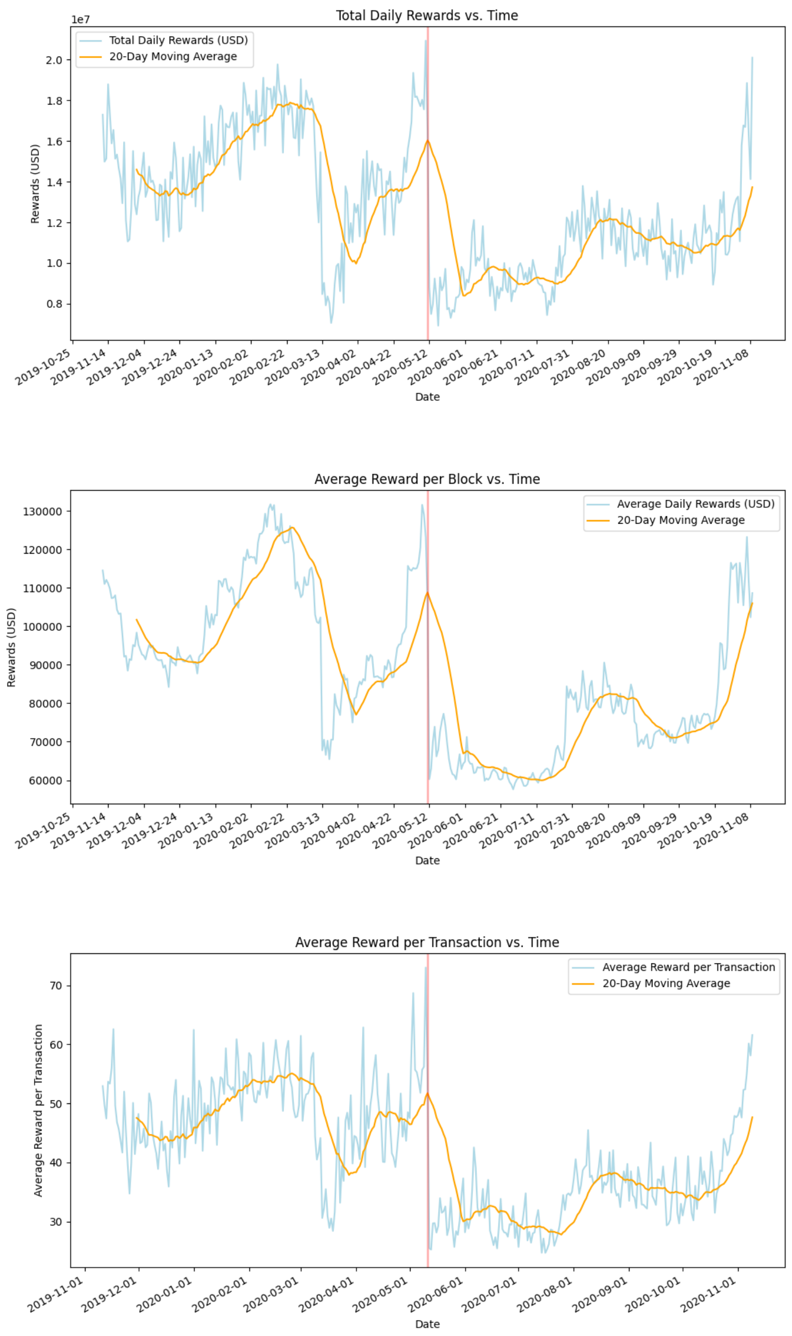


Figure 7.2. Daily summation, average per block, and average per transaction per day, of the rewards (USD) received and the twenty day moving average for each plot.

7.2.4. Difficulty. Following the visualization of rewards, we visually assessed the trends of the predictor difficulty. Difficulty is an arbitrary measurement of the computational power needed to validate a transaction. The greater the difficulty, the greater computational power needed to find the nonce. Difficulty is determined by an algorithm responding to the number of active participants. The greater the number of participants the higher the difficulty to find the nonce and validate the transaction. Figures for each predictor variable calculated from difficulty can be seen figure 6.3.

Viewing the total of daily difficulty, local minima can be observed at March and May. We reason that as the value of Bitcoin rises, the number of active participants increases, thus causing an increase in difficulty. Overall, the total difficulty can be seen to follow a similar trend to Bitcoin value.

When observing difficulty per block, a pattern consistent with discrete values is seen. Clearly this confirms the level of difficulty is determined for each block. Furthermore, the difficulty can be used an an indirect assessment of active participants in the network responding to an increase or decrease the price of Bitcoin. Given crypto currencies suddenly increase in price due to expectation of future value, commonly known as ”hype”, monitoring the difficulty can be an excellent metric to assess the overall state of the network’s price expectations.

Last, the figure 6.3 showing the trend of difficulty per transaction can be observed to posses a shallow upward trend, in correlation with bitcoin price. All three plots display a relationship to Bitcoin price without being directly effected by the halving event as observed in the plots for rewards.

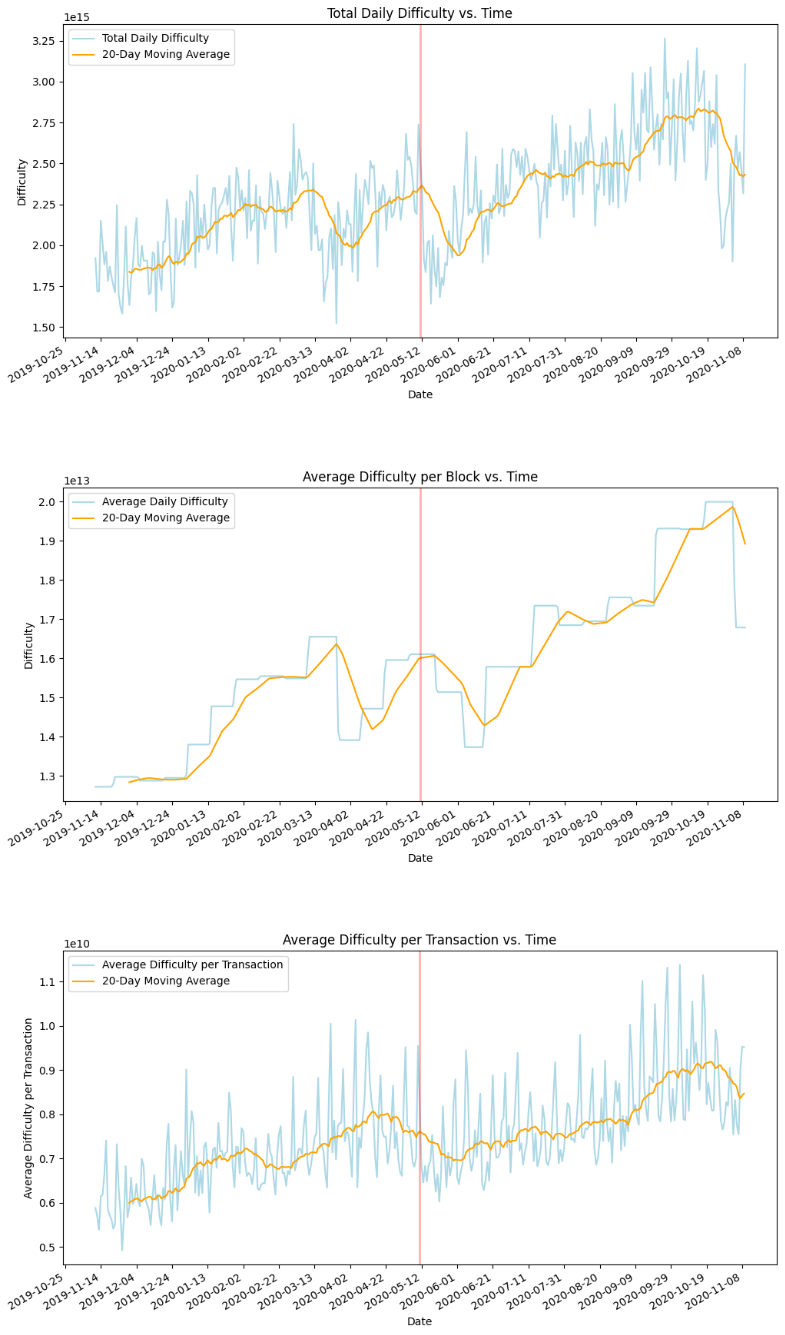


Figure 7.3. Daily summation, average per block, and average per transaction of difficulty and a twenty day moving average for each plot.

7.2.5. Transaction Fee (USD. While a fee charged by the block-chain miners for validating a transaction is similar the reward metric, the reward metric within on-chain data represent the fee charged for transaction validation, plus the reward in bitcoin for transaction validation, plus the reward for validating a block. In contrast, the fee is monetary value a participant charges not counting the additional rewards they receive for validation.

Looking at all three graphs depicted in figure 6.4 and calculated from the fee metric, nearly the exact same trend is observed for each graph. This observation can be explained by the fact the charged fee is independent from the number of blocks and total transactions validated in one day. Instead, the percent increase or decrease in transaction fees is more dependent on a minor's capacity to generate a profit. The aggregate percent change in charged fees occurs at the transaction level. Therefore, calculating the fees charged per block, or summed over a period of time should result in the same percent change.

At the halving point, the transaction fee can be seen to significantly increase to make up for lost profit due to rewards being reduced by one-half. Also, as the value of bitcoin increases, the fee charged by a minor will also increase. Still, the transaction fees appears not only correlated to Bitcoin value, but at certain dates inversely correlated to reward and difficulty. While the halving point definitively decrease rewards and difficulty, the fee is observed to increase. Upon closer inspection, small differences can be discerned between each graph depending on the scale, and hence each trend may contain information capable of contribution towards the training of a model.

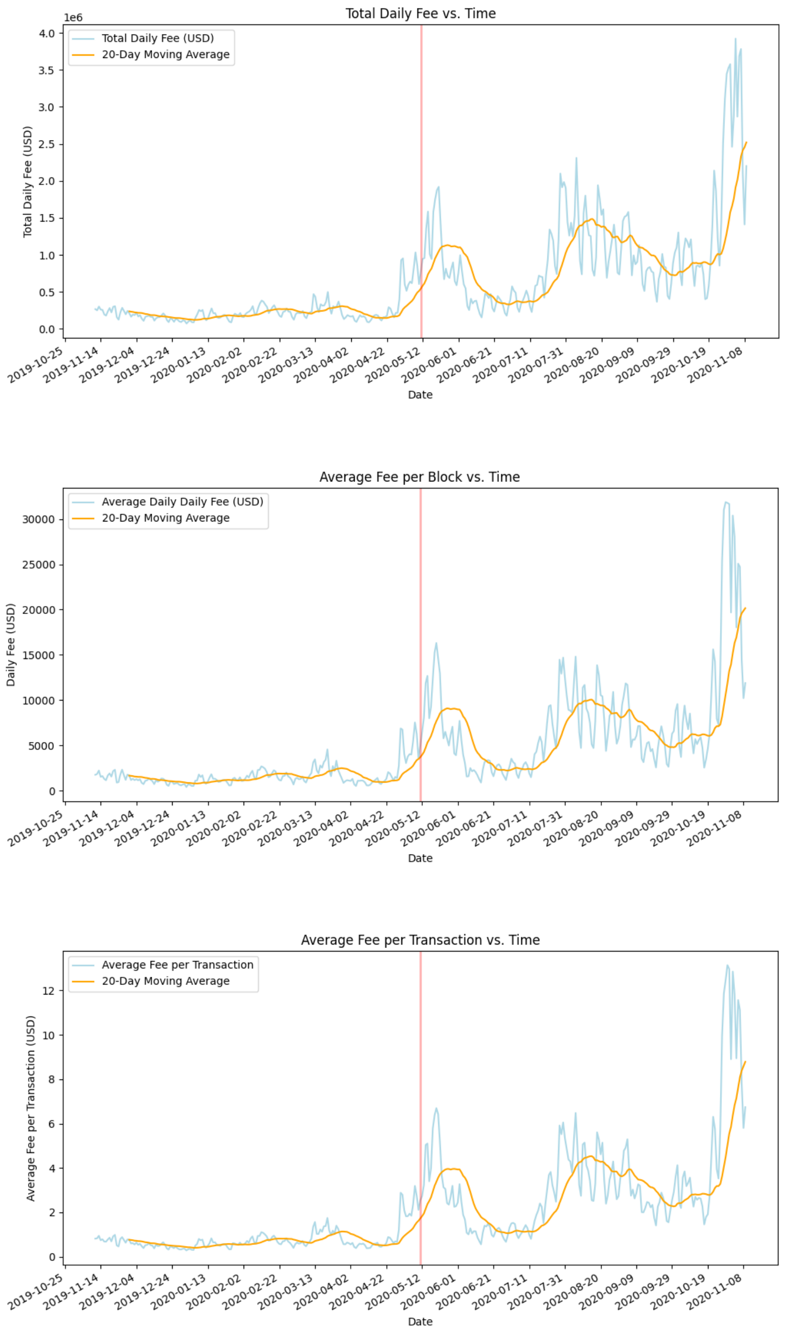


Figure 7.4. Daily summation, average per block, and average per transaction of transaction fees (USD) and a twenty day moving average for each plot.

7.2.6. Transaction Count. The last variable to be visualized was the transaction count, the total number of transaction validated within a single block. The first observation made is a clear correlation to the difficulty predictors with local minima seen to occur at the same points in time, mid April and mid June. If transactions decrease due to fewer participants within the network, it would be expected the difficulty would decrease to bring transaction validation volume back to an expected baseline. Other than the minima seen during the Bitcoin price decline in March and at the halving event in May, the volume of transaction appears to be fairly stable. This suggests transactions volumes are mostly affected by periods of high volatility during a downturn as participants hold on to assets while waiting for the price to reach is minimum value. Overall, we expected transaction fees to have a modest contribution to the prediction of Bitcoin price.

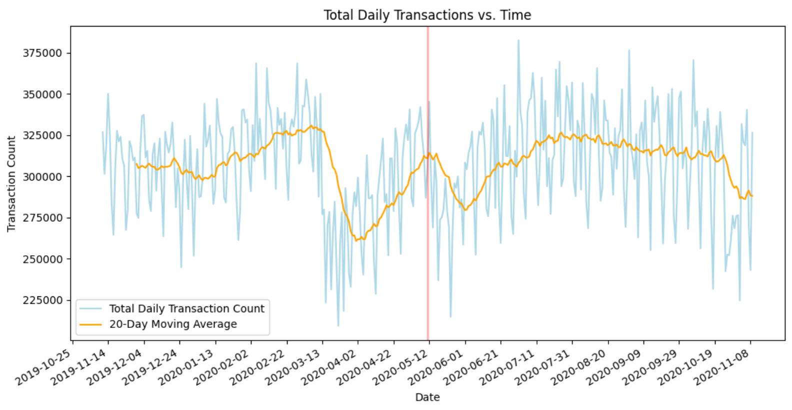


Figure 7.5. Daily summation of transaction count and a calculated twenty day moving average.

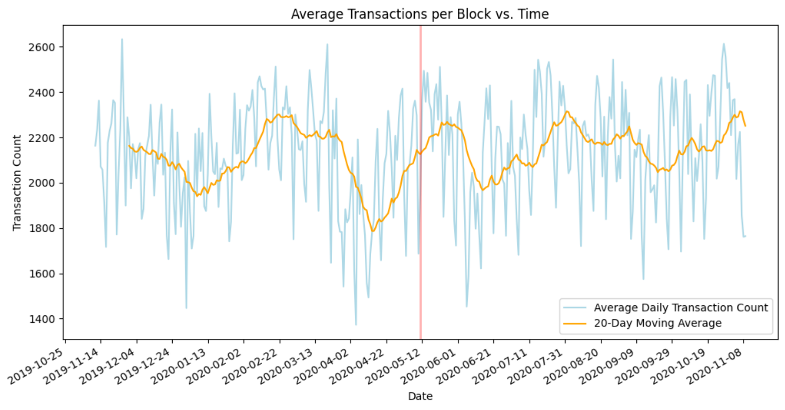
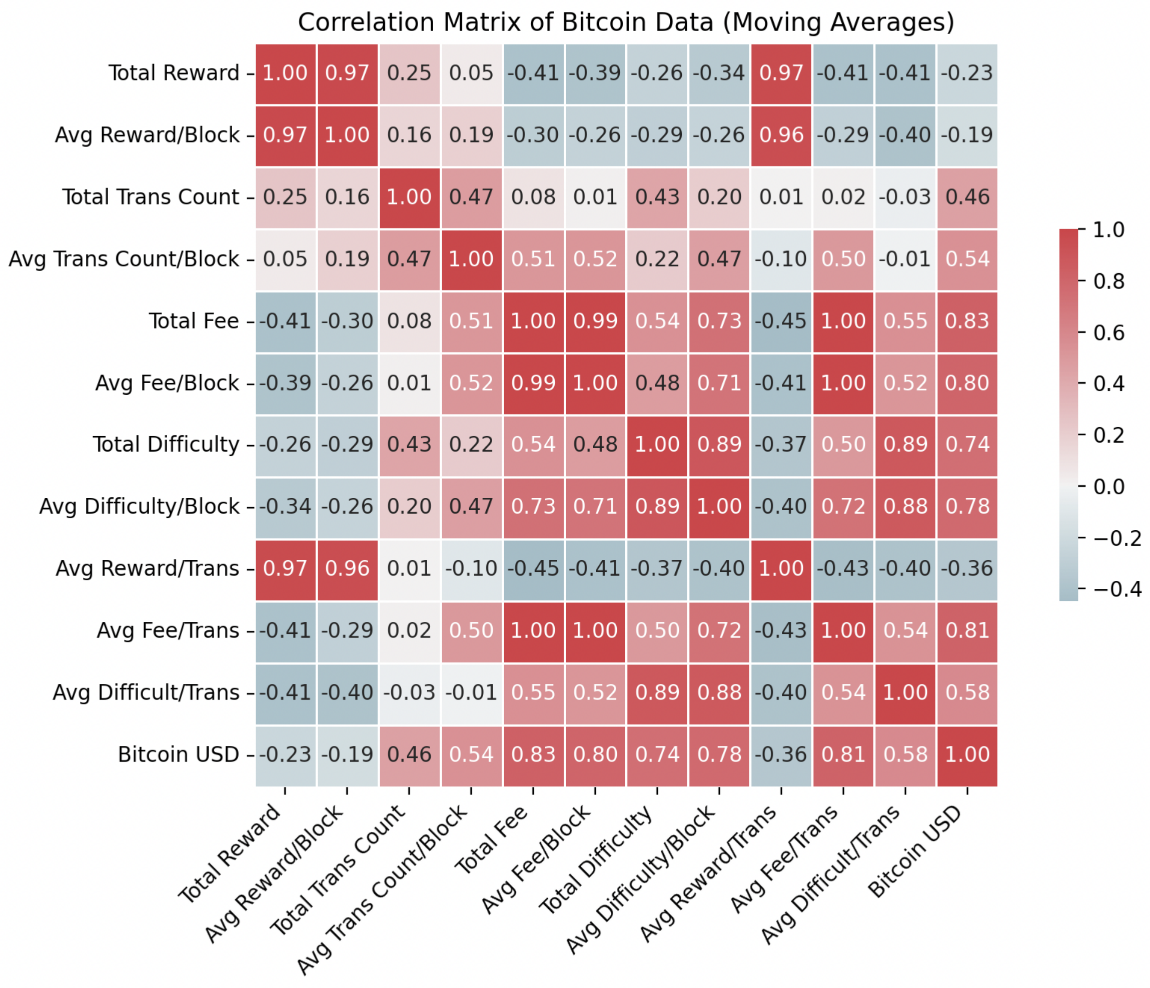


Figure 7.6. Average per block of transaction count and a calculated twenty day moving average.

7.2.7. Correlation Matrix. Following the data visualization of the predictor variables a heat map with correlation coefficients between each predictor and the price of Bitcoin were generated and is displayed in figure 6.7. Each variable in the matrix is the twenty day moving average, with the intended output for prediction being the twenty day moving average of Bitcoin price.

As expected, the variables calculated from rewards had a negative correlation between -0.19 and -0.36. These correlation values can be explained by the halving event causing a shift in the data. Not considering the halving event, rewards can be seen to closely mirror the trend of Bitcoin. Given the halving event is directly effecting the general correlation to bitcoin and artificially lowering is correlation to Bitcoin price, it was decided that it would not be incorporated in the model to predict Bitcoin price.

Looking at other variable, the total fee, summed over the date, was found to have the highest correlation to Bitcoin price at 0.83. Similarly, the average fee per block and average fee per transaction was found to correlate at 0.80 and 0.81 respectively. The similarity of correlation values is unsurprising given the trends are nearly identical. Total difficulty and average difficulty per block were found to be correlated to bitcoin price at 0.74 and 0.78



	Total Reward	Avg Reward/Block	Total Trans Count	Avg Trans Count/Block	Total Fee	Avg Fee/Block	Total Difficulty	Avg Difficulty/Block	Avg Reward/Trans	Avg Fee/Trans	Avg Difficult/Trans	Bitcoin USD
Total Reward	1.00	0.97	0.25	0.05	-0.41	-0.39	-0.26	-0.34	0.97	-0.41	-0.41	-0.23
Avg Reward/Block	0.97	1.00	0.16	0.19	-0.30	-0.26	-0.29	-0.26	0.96	-0.29	-0.40	-0.19
Total Trans Count	0.25	0.16	1.00	0.47	0.08	0.01	0.43	0.20	0.01	0.02	-0.03	0.46
Avg Trans Count/Block	0.05	0.19	0.47	1.00	0.51	0.52	0.22	0.47	-0.10	0.50	-0.01	0.54
Total Fee	-0.41	-0.30	0.08	0.51	1.00	0.99	0.54	0.73	-0.45	1.00	0.55	0.83
Avg Fee/Block	-0.39	-0.26	0.01	0.52	0.99	1.00	0.48	0.71	-0.41	1.00	0.52	0.80
Total Difficulty	-0.26	-0.29	0.43	0.22	0.54	0.48	1.00	0.89	-0.37	0.50	0.89	0.74
Avg Difficulty/Block	-0.34	-0.26	0.20	0.47	0.73	0.71	0.89	1.00	-0.40	0.72	0.88	0.78
Avg Reward/Trans	0.97	0.96	0.01	-0.10	-0.45	-0.41	-0.37	-0.40	1.00	-0.43	-0.40	-0.36
Avg Fee/Trans	-0.41	-0.29	0.02	0.50	1.00	1.00	0.50	0.72	-0.43	1.00	0.54	0.81
Avg Difficult/Trans	-0.41	-0.40	-0.03	-0.01	0.55	0.52	0.89	0.88	-0.40	0.54	1.00	0.58
Bitcoin USD	-0.23	-0.19	0.46	0.54	0.83	0.80	0.74	0.78	-0.36	0.81	0.58	1.00

Figure 7.7. A heat map of correlation values for all moving averaged variables including Bitcoin value in USD.

respectfully.The average difficulty per transaction however was found to be 0.58. Finally, predictor variables calculated from transaction count were found to correlate to bitcoin price at 0.46 and 0.54, for total transaction count per day and transaction count per block respectively.

When comparing the predictor variables to one another, difficulty and transaction fee can be seen to be correlated at approximately 0.70. Other comparison of predictors can be seen to correlate between 0.20 to 0.50. We decided to use PLS regression to account for the moderate to high correlation between predictor variables. To generate the model, all predictors were decided to be incorporated except for variables based on the rewards, leaving a total of eight variables of moving averages. While it is highly likely a subset of the selected predictors would be sufficient, the goal of analysis was to generated a model that predicts the price of bitcoin with a higher level of accuracy than simply using a single predictor.

7.3. PLS REGRESSION

7.3.1. Component Analysis. Before the analyses was ran, all predictor variable were standardized with K-fold cross validation utilized for determining an estimate of the error. With 366 - 19 = 347 data points, the data was randomised then split into five spaces with each possible combination of components and predictor variable used to generated a data frame displaying the RMSE for all combinations. From the output, using all eight predictors with eight components provided the lowest RMSE. While other combinations might equally effective at prediction, it was decided to move forward with eight predictors and save model selection for future work.

With eight predictors yielding the lowest RMSE , the individual RMSE was calculated each for one to eight components to determine the appropriate number of components to analyze. This value was found by determining the RMSE for each fold used in a testing set, then the average of all five RMSE values found over each round was calculated. In addition, a plot of the percent variance of the output, Bitcoin value, was generated in relation to the number of components used.

From both the table 6.2 and figure 6.8, a total of seven components was shown to be the optimal number for use. The RMSE and the graph explaining the variance of the output can each been seen to level off in their respective measured values when seven components is reach. After deciding the number of component to utilize in the model, data analysis was conducted to determine the error of the regression, the linear equation, and weights of the components.

Table 7.2. Root mean square for each additional component implemented into the model up to eight predictor variables.

Principal Component	RMSE
1	709.7
2	617.3
3	563.7
4	556.7
5	521.4
6	498.0
7	482.9
8	482.3

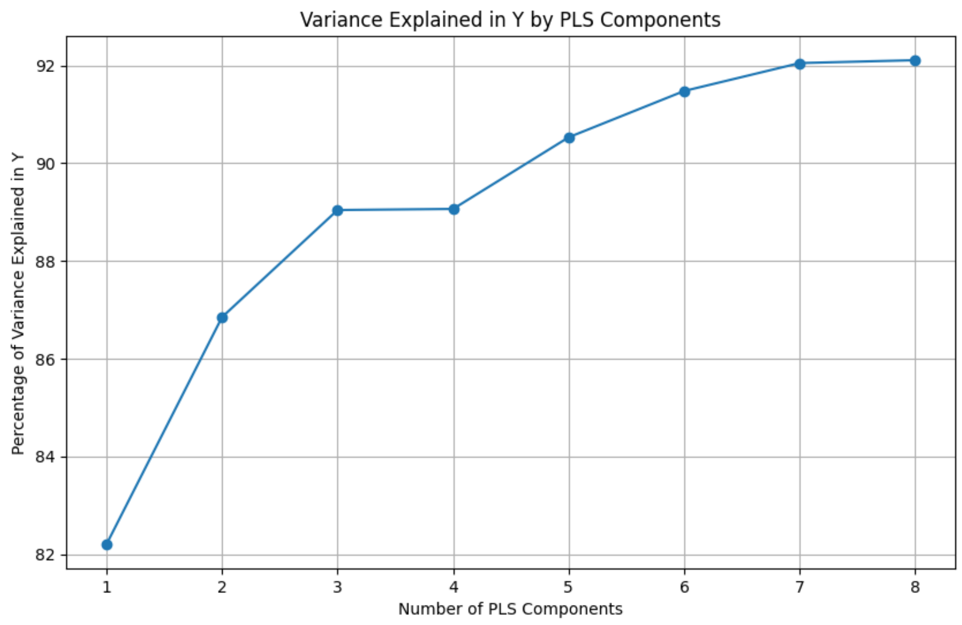


Figure 7.8. A plot of the variance explained in the output Y, Bitcoin value (USD), in relation to the number of components used in the PLS regression.

7.3.2. PLS Data Analysis. To begin the data analysis of the fitted regression using PLS, the linear equation for the PLS regression was obtained when using the latent variables, displayed in table 6.3. While the linear equation is valuable for understanding the latent variable contribution to the regression, no valuable insight into the predictors contribution to the equation can be found. This lack of insight is due to the fact that latent variables are computed from the predictor variables. The following table provides the coefficients for each latent variable.

Along with the linear equation, the weights of each predictor and their respective component were also obtained. From the weights displayed in table 6.4, within the first component, predictors based upon transaction fees have a largest contribution at approximately 0.400. Following close behind are predictors based on difficulty at approximately 0.390. Predictors found using transaction count have contribution at approximately 0.250. Each predictor also exhibits a positive contribution within the first component. It should be noted that predictors based upon the same variable exhibit similar levels of weights suggesting that model reduction is possible with further experimentation. The second component was found to be fairly dominated by total transaction count, while the third component was found to be fairly dominated transaction count per block.

Table 7.3. Latent variables implemented in the PLS regression and their coefficients in the linear equation.

Latent Variable	Coefficient
z_1	1534.325
z_2	364.754
z_3	-250.439
z_4	-25.521
z_5	-204.830
z_6	164.713
z_7	127.867

Table 7.4. Predictor variables implemented in the PLS regression and their wieghts for components 1 through 7

Predictor Variable	Component 1	Component 2	Component 3	Component 4	Component 5	Component 6	Component 7
Total Transaction Count	0.232	0.881	-0.013	0.150	-0.182	-0.040	0.234
Transaction Count per Block	0.269	0.067	0.837	-0.266	0.176	0.111	-0.297
Total Fee	0.417	-0.008	-0.224	0.120	0.858	-0.068	0.140
Fee per Block	0.402	-0.048	-0.250	-0.494	-0.207	-0.650	-0.247
Total Difficulty	0.372	0.076	-0.055	0.504	-0.143	0.075	-0.558
Difficulty per Block	0.389	-0.274	0.318	0.164	-0.240	-0.193	0.669
Fee per Transaction	0.405	-0.038	-0.281	-0.452	-0.163	0.712	0.099
Difficulty per Transaction	0.290	-0.366	-0.060	0.405	-0.230	0.096	-0.090

After obtaining the linear equation and evaluating the weights of the components, various metrics were calculated to assess the error of the PLS prediction model using K-fold cross validation for each metric. Similar to the previous cross validation, data was randomized and the number of folds implemented was five. Each result from the testing set was then combined to an overall evaluation relative to the original data. The first metric obtained was the RMSE, found to equal 486.38. The RMSE value provides an average distance of 486.38 USD from the predicted output to the true value. The training model was found to have an RMSE at 463.59, two relatively close values. Next, the cross validated R^2 value was found to equal 0.92. Likewise, the R^2 value found using the training set was also found to equal 0.92. The closeness of these two values suggests PLS regression has a predictive goodness of fit equivalent to the trained data. Finally, a plot of the fitted PLS regression against the moving average of Bitcoin value and a plot of the predicted Bitcoin price against the moving average of Bitcoin value was generated.

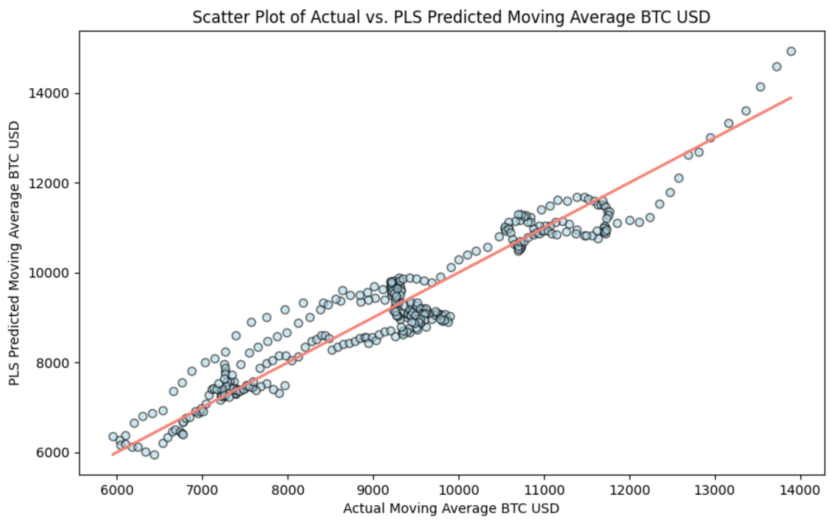


Figure 7.9. The fitted PLS regression flattened to a one dimensional array and plotted against the 20-day moving average value of Bitcoin is USD.

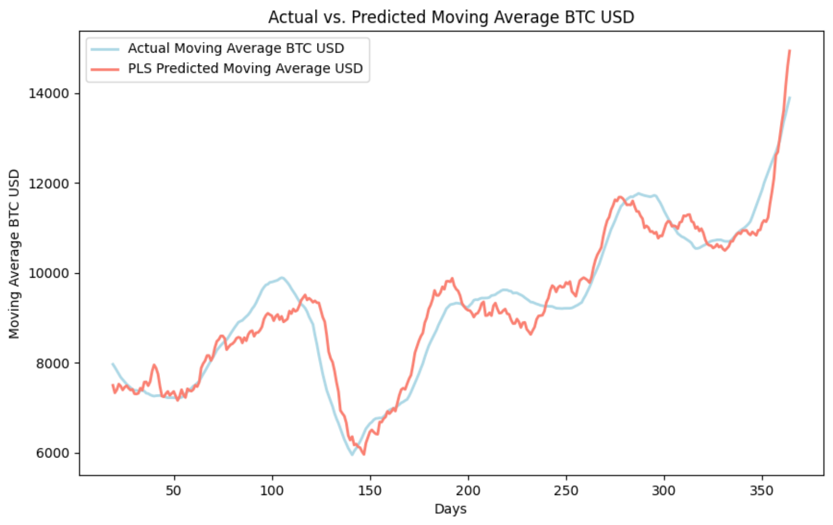


Figure 7.10. The predicted value of Bitcoin price (USD) derived from the fitted model using PLS regression compared to the true price relative to the time in days.

As observed in figure 6.9, the fitted line appears to have a strong fit to the moving average Bitcoin value. Likewise in figure 6.10, the predicted value closely resembles the true curve of the moving average Bitcoin price. It can also be observed that the predicted price tends to underestimate the true price before a decline in value while also exceeding the true price before an increase in value. This observation can been seen approximately occurring at days 75, 180, 225, and 280 suggesting PLS regression may be able to help anticipate moving average price movement with further exploration and refinement.

8. CONCLUSION

From the results obtained from the study, we were successfully able to use metrics from on-chain data to predict the moving average value of Bitcoin. On average, the distance form the predicted value to the true moving average value was 486 US dollars. In addition, correlation to the bitcoin value was improved to a value 0.92. We believe that implementing such a model to predict moving averages of crypto currencies could be useful in developing trading strategies. All crypto currencies include on-chain data and similar techniques may be applied to a wide variety of on-chain data.

Future work that could expand upon the model obtained in the this study includes model reduction, expansion of time period analyzed, and incorporating other on-chain data metrics. While the model form this study used eight predictor variable, we believe that a simpler model with adequate predictive capacity can be obtained. Trends using transaction fee as the base variable appear nearly identical. Incorporating only one of the three predictors may suffice. Also, the time period in which to train the data can also be expanded to exclude the halving event. Exclusion of the halving event may allow predictors based on rewards to be incorporated, given its trend closely resemble Bitcoin value with the exception of the halving event. In addition, using a moving PLS regression may be interesting to avoid the use of outdated correlations. Finally, reducing the time period for the moving average may yield an advantage to predict the direction of movement.

Overall, the data analysis provided a successful prediction of Bitcoin's moving average price. While there is room for improvement, utilizing correlated on-chain data to generate a predictive model using PLS regression appears to show promise as an analytical technique for price and movement prediction. Utilizing this methodology on other crypto currencies such as Ethereum that include addition on-chain metrics have the potential to yield similar results.

REFERENCES

[1] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015. URL https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[2] Einar Mykletun. Maithili Narasimha. Gene Tsudik. Providing authentication and integrity in outsourced databases using merkle hash trees. URL https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkleodb.pdf.

[3] Andrey Sergeenkov. What is an automated market maker? 2023. URL https://www.coindesk.com/learn/what-is-an-automated-market-maker/.

[4] Tokenobu Tani. Ethereum evm illustrated. 2017. URL https://github.com/takenobu-hs/ethereum-evm-illustrated?tab=readme-ov-file.

[5] Corwin Smith. Transactions. 2024. URL https://github.com/takenobu-hs/ethereum-evm-illustrated?tab=readme-ov-file.

[6] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger paris version. 2024. URL https://ethereum.github.io/yellowpaper/paper.pdf.

[7] Harold Deal. Alic Droogan. Cynthia Fuller. Working draft american national standard x9.62-1998 public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ecdsa). 1998. URL https://safecurves.cr.yp.to/grouper.ieee.org/groups/1363/private/x9-62-09-20-98.pdf.

[8] Donald Johnson. Alfred Menezes. Scott A. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1:36–63, 2001. URL https://link.springer.com/article/10.1007/s102070100002.

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

[10] Pablo Pettinari. Ethereum virtual machine. 2023. URL https://ethereum.org/en/developers/docs/evm.

[11] Nico. Introduction to smart contract. 2023. URL https://ethereum.org/developers/docs/smart-contracts.

[12] Nico. Gas and fees. 2023. URL https://ethereum.org/developers/docs/gas.

[13] Corwin Smith. Smart contract languages. 2024. URL https://ethereum.org/developers/docs/smart-contracts/languages.

[14] Corwin Smith. Transactions. 2024. URL https://ethereum.org/developers/docs/transactions.

[15] Nico. Ethereum accounts. 2024. URL https://ethereum.org/developers/docs/accounts.

[16] Erc-20 token standard. 2024. URL https://ethereum.org/developers/docs/standards/tokens/erc-20.

[17] Ori Pomerantz. Uniswap v2 contract walk through. 2021. URL https://ethereum.org/en/developers/tutorials/uniswap-v2-annotated-code.

[18] Hayden Adams. Noah Zinsmeister. Dan Robinson. Uniswap v2 core. 2020. URL https://uniswap.org/whitepaper.pdf.

[19] Trevor Hastie. Robert Tibshirani. Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2017.

[20] Herve Abdi. Partial least squares (pls) regression. URL https://personal.utdallas.edu/~herve/Abdi-PLS-pretty.pdf.

[21] Artur Meynkhard. Fair market value of bitcoin: Halving effect. 2019. URL https://pdfs.semanticscholar.org/3686/3ee36a6e246b23c5baf2ba1e623ffd1fa832.pdf.

VITA

Paul Kenneth O’Connor originally obtained a Bachelor’s of Arts from Augustana College in Philosophy in 2009. Following this degree a Bachelor’s of Science was obtained in Chemistry from Saint Ambrose University in 2019. After obtaining this degree, he found employment within industry and was promoted to Quality Engineering in 2022. While employed as an engineer responsible for data analyses of quality processes, a Master’s of Science was obtained in Applied Mathematics from Missouri University of Science and Technology in 2024.