CAPTURING THE PREDICTIVE POWER OF CORTICAL LEARNING ALGORITHMS

By

Alexander C Michels

Submitted to the Department of Computer Science and Mathematics

Westminster College, New Wilmington, PA

In partial fulfillment of the requirements for Honors Research

advised by

Carolyn Cuff, Ph.D.

C. David Shaffer, Ph.D.

William Procasky, Ph.D.

April 29, 2019

Contents

ACKNOWLEDGEMENTS

I would like to thank my Honors Board: Dr. Carolyn Cuff, Dr. C. David Shaffer, and Dr. William Procasky for their guidance throughout this project and over the past few years. Not only were they instrumental to this thesis and incredible professors through my time at Westminster, they were my advisors in Mathematics, Computer Science, and Financial Economics respectively.

I cannot thank the Mathematics and Computer Science faculty at Westminster College enough for everything they have done for me. They changed my perspective on mathematics and computer science, gave me ample opportunities to explore topics that interested in me, and were instrumental in many of the opportunities I have received.

Finally, I would like to thank Marcus Lewis and Scott Purdy at Numenta for taking the time to discuss Hierarchical Temporal Memory and my research questions with them.

ABSTRACT

Hierarchical Temporal Memory (HTM) is model of intelligence based on the the interactions of pyramidal neurons in the mammalian neocortex which is currently being developed by Numenta. It has stood out due to high noise tolerance and learning capacity which makes it well-suited for anomaly detection, prediction, signal processing, and sensorimotor processes. We seek to find a mathematical analogy to the predictive power of this biologically constrained theory using models from time series analysis. To do this, we statistically analyzed the predictions of HTM networks which were asked to predict outputs of autoregressive moving average (ARMA) models, varying the parameters of both the networks and the ARMA models. We hope to to find a relation between sets of HTM network parameters and ARMA model parameters to better explain cortical learning algorithms and the neocortex.

List of Figures

List of Tables

Chapter 1

Introduction

Although we are continually bombarded with sensationalist news stories proclaiming the dawn and dangers of "artificial intelligence," it is important to first define what it means to say a machine is intelligent. This is much the same way Turing began his preponderance of artificial intelligence in his 1950 *Computing Machinery and Intelligence* [34]. Turing came up with an eloquent boundary for determining the point at which we call a machine 'intelligent.' His proposal, which he called "The Imitation Game," but is now referred to as "The Turing Test," is to have an interrogator question a human and an artificial intelligence, randomly labeled X and Y, in the hopes of distinguishing between the two [34]. We reach "intelligence" when an interrogator cannot reliably distinguish between the two [34].

As mathematicians, we could also take our favorite route of defining something: we look at a collection of things that we would agree to be "intelligent" and abstract the shared properties we consider to be desirable until we have a set of properties that must be met for a system to be considered intelligent. From this approach, we could say that a system **S** is intelligent if and only if it is able to "use language,

form abstractions and concepts, solves kinds of problems now reserved for humans, and improve themselves" [24]. This is the definition from the Dartmouth AI Summer Research Project, but such a definition is obviously hard to evaluate and it would be much more difficult to form a consensus on what set of properties define intelligence than it is to get a set of properties to define other abstract mathematical concepts such as an integral domain.

It quickly becomes apparent that we also need more states than just the two Boolean "intelligent"/"not intelligent" ones to talk about intelligence in a constructive manner. Without this everything on the spectrum from a for loop with an if-else to the robots in Isaac Asimov's *I, Robot* are under the same label of "not intelligent" yet we know that the 'intelligence' exhibited in those cases are not at all comparable. We need a spectrum with many states of intelligence to constructively talk about the intelligence of a system.

John Searle's 1980 paper "Minds, Brains, and Programs" introduced the world to The Chinese room argument [33]. It supposes that artificial intelligence research is successful and produces an artificial intelligence that is capable of behaving as if it understands Chinese, then asks does the machine literally understand Chinese or it simulating the ability to understand Chinese? [33]

Figure 1.1: Searle's Chinese Room[1]

Although this may seem like a pedantic distinction at first glance, the difference is truly important. The hypothesis that we can only ever simulate the ability to think is known as the weak AI hypothesis whereas the hypothesis that we can produce a machine capable of thought is the strong AI hypothesis [24]. These both stand in contrast to Artificial General Intelligence, which is an intelligence capable of performing any intellectual task that a human can [7].

1.1 The History of Artificial Intelligence

One would think that artificial intelligence would have its roots in the last century or two, but mankind has dreamed of and proposed machines with human-like intelligence for thousands years, dating back to at least Homer [7]. From our imagination and literature, artificial intelligence was brought into the realm of the academic by philosophers and mathematicians such as René Descartes' "mechanical man" and Gottfried Wilhelm Leibniz's mechanical reasoning devices [7]. Pascal and Leibniz both designed calculating machines capable of automated arithmetic, but proposing a calculator is far from what we think of as artificial intelligence today [7].

It was the rise of electronics from Turing, IBM, Bell Laboratories, and countless others in the mid-twentieth century started to change the question from a philosophical one to a practical one [7]. Artificial intelligence is a testament to the kind of interdisciplinary problem solving encouraged by the liberal arts, with contributions coming from the fields such as engineering, biology, psychology, game theory, communication theory, logic, philosophy, and linguistics [7]. Eventually, with advancements in computational power, operating systems, and language design, researchers

[1]Image: https://theness.com/neurologicablog/index.php/ai-and-the-chinese-room-argument/

were able to demonstrate impressive computational problem solving such as Arthur Samuel's 1952 checker-playing program written in assembly language and one of the first examples of evolutionary computation [7].

Newell, Shaw, and Simon's "Logic Theorist" program became the first artificial intelligence written for a computer in 1956 [18]. Through the use of heuristic algorithms, Logic Theorist was able to prove theorems and solve complex problems astonishingly well [18]. These early attempts at artificial intelligence were largely doing two things: searching and finding ways to represent and manipulate knowledge. Claude Shannon pointed this out in his 1950 "Programming a Computer for Playing Chess" in which he produced what is now called the Shannon number, 10^{120}, which is a lower bound of the game tree complexity of Chess [24].

Artificial intelligence became formally recognized as a field of study and got its name from the 1956 Dartmouth Artificial Intelligence Conference [7]. Another product of the conference was a step forward in artificial intelligence's ability to represent and manipulate knowledge with John McCarthy's development of the first AI programming language, LISP [24]. The strides towards strong AI came crashing down with the publication of the 1969 paper "Perceptrons" which showed that single layer perceptrons were not able to properly handle linearly inseparable problems, leading to a steep decline in neural network research and "AI Winter" [24].

Research into artificial intelligence reemerged in the mid to late eighties, but this time with a more practical focus rather than searching for Searle's strong AI [24]. Algorithms developed and used for artificial intelligence found their way into camera auto-focus, anti-lock brakes, search engines, and medical diagnoses [24]. Another marked difference is the plethora of approaches such as agent systems and biologically inspired systems. Today research into artificial intelligence has largely remained in

this practical realm, using neural networks, data mining, fuzzy logic and other tools to solve real-world problems while slowly marching towards an Artificial General Intelligence.

1.2 Approaches to Intelligence

Artificial intelligence, because of how broadly the word can be defined and how many fields contribute the its progress, can be hard to wrap one's head around. However, Connell and Livingston have proposed four categories for artificial intelligence approaches which are useful for understanding the state of artificial intelligence research and the varied potential paths to Artificial General Intelligence [9].

Their first category is labeled "Silver Bullets" and describes approaches in which much of what is needed is already believed to be present, but we are missing a crucial piece that will supposedly resolve our problems and deliver us a system with intelligence [9]. Examples include 'Fancy Logic' (second-order, non-monotonic, epistemic, deontic, modal, etc), Fuzzy Logic, Deep Language, Embodiment, and Quantum Computing [9]. Disciples of this school of thought are chasing their particular "Silver Bullet," working to formalize and perfect what they believe to be the missing link.

They next describe the "Core Values" section which puts emphasis on the central organizational scheme over other computational details, believing that this macro-level structure has greater influence than the exact algorithms used [9]. Situatedness, Emotionality, Self-Awareness, and Hierarchy & Recursion are a few of these ideologies. There are strong arguments for this category, especially Hierarchy & Recursions argument that an intelligence needs to be able to abstract recursively [9].

Connell and Livingston's third category, "Emergence," looks at artificial intel-

ligence approaches which believe they already have the essentials, but we haven't implemented the essentials on a large enough scale to get our intelligence yet [9]. For example, one might hold the position that intelligence is simply the ability to generate and search decision trees and we haven't realized an Artificial General Intelligence yet because our hardware doesn't allow us to do this effectively enough yet. Approaches in this category include Axiomatization, Commonsense, Learning, Evolution, and Integration [9].

Lastly, we visit "Emulation" which is the school of thought that says we are better off copying intelligence than designing our own [9]. Neural simulation, neural networks, animal models, human development, sociality, and cortical learning algorithms all fit in this category [9]. The danger with this approach is abandoning theory in its sprint towards a functional copy, because if one does not understand the thing they have made, it is hard to see what it can do and where it can be improved. Another excellent point is that it can be very hard to correctly identify what needs to be copied, as Connell and Livingston note, "artificial feathers and flapping turn out not to be needed to create airplanes" [9].

1.3 Computational Neuroscience

Emulating natural processes–and even the human brain–is not a new idea to computer scientists. In fact, artificial neural networks that we still use today were originally proposed in a 1943 paper that introduced the first mathematical model of a neuron [12]. The 1950's and 60's saw the rise of evolutionary computing which attempts to simulate evolution usually for finding optimal solutions to problems. Computational neuroscience is the newest field to emerge from this interdisciplinary

intersection of biologists and computer scientists.

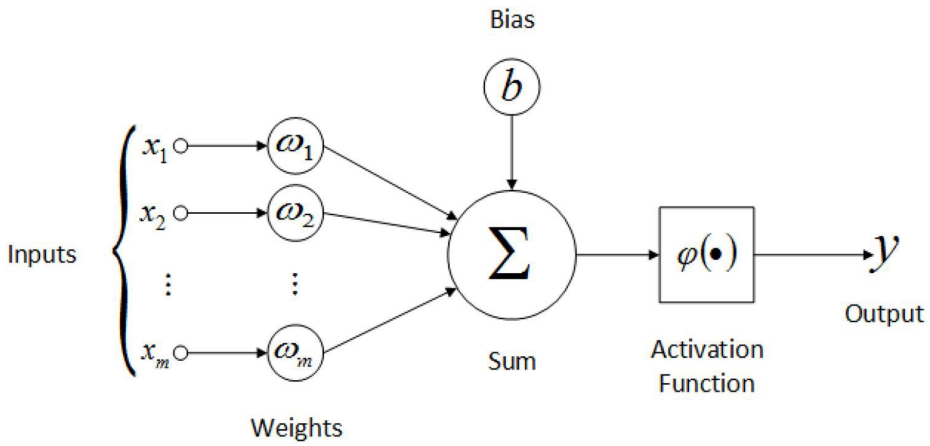


Figure 1.2: A Model for an Artificial Neuron[2]

Computational neuroscience can be described as "simulating human cognition using biologically based networks of neuronlike units" [14]. In other words, the field has tasked itself with the incredibly difficult task of trying to understand cognition by breaking the brain down into algorithms and data structures which we can then implement in a computational model. So rather than trying to create a computational model that is as intelligent as the brain, they are taking the more direct approach in attempting to model the brain itself.

One such set of computational neuroscientists are the researchers at Numenta who are attempting to replicate the mammalian neocortex. Their founder, Jeff Hawkins, realized in the early 1980's that we needed to truly understand what made the brain intelligent before we created our own intelligence [23]. Unfortunately, this view was not shared by researchers at Intel and MIT where he unsuccessfully attempted to start a research lab and study computer science respectively. To support his inquiry,

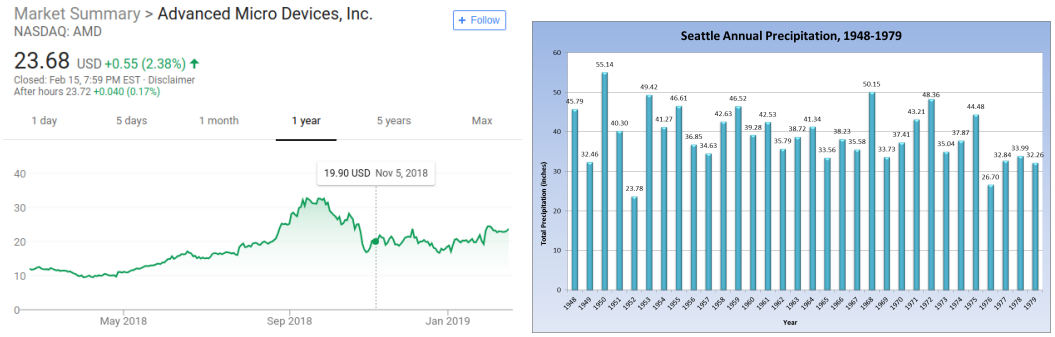
[2]Image: https://en.wikipedia.org/wiki/Artificial_neural_network

Jeff Hawkins decided to supplement his computer science and engineering background with some training in biophysics and set out to develop a theoretical understanding of the neocortex. In 2002, he founded the Redwood Neuroscience Institute and Hawkins is currently developing cortical learning algorithms and Hierarchical Temporal Memory at Numenta which he founded in 2005 [23].

Chapter 2

Time Series

A **time series** is a "sequence of observations taken sequentially in time" and are everywhere in today's world [6]. Figure 2.1 shows two such examples, stock prices (Figure 2.1a) and levels of precipitation in Seatle, WA from 1948-1979 (Figure 2.1b). Their ubiquity drives a desire to understand, characterize, and predict time series, but it can be an extremely complex task because there is generally a dependence on prior terms due to the sequential nature and some uncertainty to their movements.



(a) A chart of AMD stock prices (b) Seattle precipitation, 1948-1979

Figure 2.1: Examples of Time Series [1]

2.1 Autoregressive Models

One model for representing time series is the autoregressive (or AR) model. In an autoregressive model, the series at some time t is expressed as a finite weighted sum of previous values of the series plus a random component a_t [6]. Let $z_t, z_{t-1}, z_{t-2}, ...$ denote the series at times $t, t-1, t-2, ...$ Further, let μ be the mean or level of the process and let $\tilde{z}_{t-n} = z_{t-n} - \mu$. We can then express an autoregressive process of order p by

$$\tilde{z}_t = \phi_1 \tilde{z}_{t-1} + \phi_2 \tilde{z}_{t-2} + ... + \phi_p \tilde{z}_{t-p} + a_t \tag{2.1}$$

where $\mu, \phi_1, \phi_2, ..., \phi_p, \sigma_a^2$ are the $p+2$ unknown parameters of the model with σ_a^2 being the variance of the white noise process a_t [6]. Using a backshift operator B such that $Bz_t = z_{t-1}$ we can define an autoregressive operator of order p by

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - ... - \phi_p B^p \tag{2.2}$$

which allows us to write an autoregressive model succinctly as $\phi(B)\tilde{z}_t = a_t$ [6].

2.2 Moving-Average Models

Suppose instead that the true process is regressed on the random shocks rather than the deviations from the level. Starting from our autoregressive model of order p we can recursively substitute the model's definition of $\tilde{z}_{t-n}$ until we end up with an infinite summation of innovation terms [6]. However, the moving average model

[1] Image (b) courtesy of Seattle Weather Blog: http://www.seattleweatherblog.com/rain-stats/

provides us a much more direct approach by representing the time series as a finite number of shocks to the process. Formally we express a moving average (MA) model of order q by

$$\tilde{z}_t = a_t - \theta_1 a_{t-1} - \theta_2 a_{t-2} - ... - \theta_q a_{t-q} \tag{2.3}$$

just as with the autoregressive model, we have $q + 2$ unknown paramters and using the backshift operator we can define a moving average operator that allows us to express the model more succinctly yielding $\tilde{z}_t = \theta(B)a_t$ [6].

2.3 Autoregressive Moving-Average Models

Real world processes may require a combination of these two approaches for a succinct and accurate model, resulting in a mixed Autoregressive Moving-Average (ARMA) model [6]. Such a model includes terms from both the autoregressive and moving average model resulting in:

$$\tilde{z}_t = \phi_1 \tilde{z}_{t-1} + ... + \phi_p \tilde{z}_{t-p} + a_t - \theta_1 a_{t-1} - ... - \theta_q a_{t-q} \tag{2.4}$$

This model has $p+q+2$ unknown parameters and can also be written more succinctly using our autoregressive operator and moving average operator resulting in $\phi(B)\tilde{z}_t = \theta(B)a_t$ [6].

2.4 Classification of Time Series

A key question in time series analysis is how to select the correct model for a given time series. There are a few approaches, but one is to use information criterion such as

the Akaike Information Criterion (AIC) or the Bayes Information Criterion (BIC) [6]. For both information criterion, optimal models minimize the criterion and there is a penalty for the number of parameters used resulting in a bias towards a smaller set of parameters. The formula for BIC is

$$BIC_{p,q} = \ln(\hat{\sigma}_a^2) + r\frac{\ln(n)}{n} \tag{2.5}$$

where n is the sample size, $\hat{\sigma}_a^2$ is the maximum likelihood estimate of σ_a^2, and $r = p + q + 1$ is the number of parameters estimated in the model. Once the sequence is classified, full identification is achieved by fitting coefficients to the lag polynomial of the model to arrive at an exact model for the given time series

2.5 Noise

In both Autoregressive and Moving-Average models there was parameter σ_a^2, which represented the variance of the white noise process a_t. This parameter, the variance of the "random shocks to the system" or "white noise processes," represents the level of randomness in the process. In real-world processes white noise can be introduced through a variety of means such as measurement error and atmospheric interference which can lead to random errors in a signal such as a radio broadcast (or "static"). Models with larger σ_a^2 will be "noisier" meaning the sequence it is modeling has more randomness in its values than a model with a lower σ_a^2.

The ubiquity of time series makes forecasting them an important and many times profitable task such as in the case of financial time series (as seen in Figure 2.1a), making them a common prediction task for machine learning and artificial intelligence

systems. It allows us to test an "intelligent" systems ability to discover temporal patterns which may sometimes be fuzzy or complex: a hallmark of human intelligence.

Chapter 3

Biologically Inspired Computing

3.1 Particle Swarm Optimization

Introduced in by Russell Eberhart and James Kennedy in their 1995 paper, *A New Optimizer Using Particle Swarm Theory*, Particle Swarm Optimization has changed the way the world optimizes continuous nonlinear functions [5]. Swarming is fast and effective for optimization over complex multidimensional search spaces and lends itself well to parallelization due to its multi-agent approach, making it an excellent choice for optimizing parameters in neural networks and is implemented by Numenta for finding optimal parameters.

Usually abbreviated to PSO or *swarming*, it draws inspiration from bird flocks and schools of fish [14]. The concept is quite simple: it simulates particles on parameter space and at each time-step every particle evaluates fitness as a function of its location in parameter space, then moves towards some linear combination of its personal best and the overall best score among the particles using weights representing a cognitive constant or individuality (denoted c) and a social constant (denoted s).

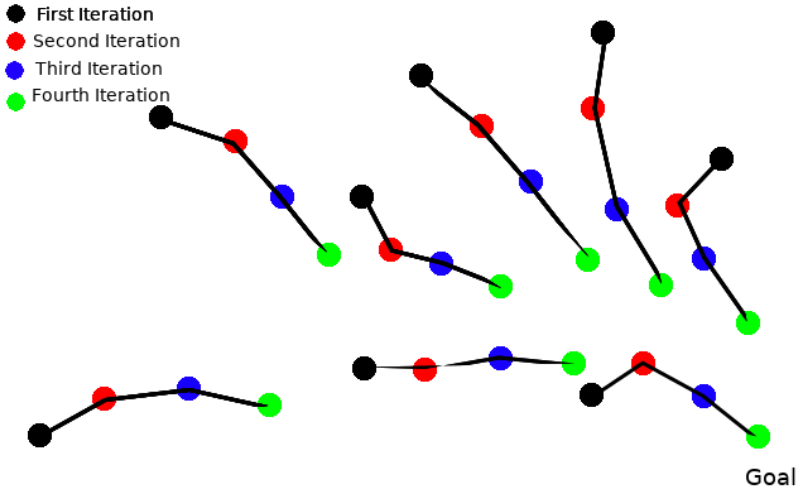


Figure 3.1: Visualization of the PSO Algorithm

The algorithm begins by initializing all particles with a random coordinate in the parameter space and a random velocity. Then at each time step, it calculates its fitness based on the function being optimized and the parameters associated with its position in parameter space. If this fitness is the best the particle has achieved, it is recorded as its personal best, denoted `pbest` and the particle with the best fitness is chosen as global best, denoted `gbest`. We then update the velocity and particle position of each particle in our swarm using the equations described in Equation (3.1) where v_t is the velocity of a particle at time step t, p_t is the position of a particle at time step t, and β_1, β_2 are uniformly distributed random variables with $\max(\beta)$ being a parameter of the algorithm [5].

$$\begin{aligned} v_{t+1} &= v_t + c \times \beta_1 \times (\text{pbest} - p_t) + s \times \beta_2 \times (\text{gbest} - p_t), \\ p_{t+1} &= p_t + v_{t+1}. \end{aligned} \tag{3.1}$$

We utilized swarming to find optimal or near-optimal sets of parameters for learn-

ing ARMA time series with a Hierarchical Temporal Memory network. Numenta themselves utilizes swarming for this very same purpose, and NuPiC comes with a swarming utility, however we wrote my own swarming utility in Python which we thoroughly tested on a number of complex surfaces.

3.2 Hierarchical Temporal Memory

Hierarchical Temporal Memory is a framework attempting to produce cortical learning algorithms—algorithms that mimic the processes of the human neocortex. In the human brain, the neocortex is approximately seventy-percent of the brain's volume [29]. It can be described as a "wrinkly sheet of sheet of cells, about two millimeters thick" and is responsible for the characteristics of intelligence that set humanity apart from other species [29].

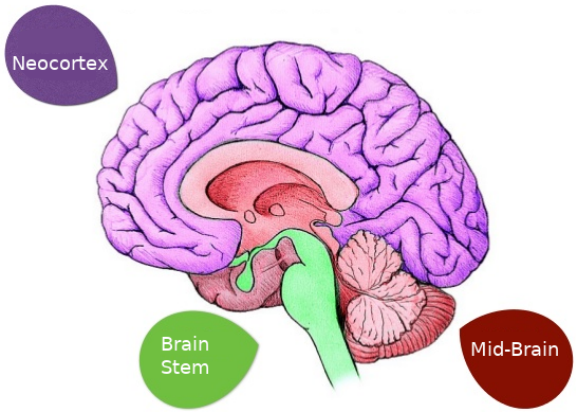


Figure 3.2: Overview of the Human Brain

For all of its complexity, the neocortex has an incredibly consistent microarchitecture of cortical columns which are in turn made up of neurons [29]. Neuroscientist Vernon Mountcastle revolutionized the way we think about our own brains when he

proposed that the consistent architecture must imply consistent processes across the cortex. This would mean that all of our high-level thinking must essentially be the same on the level of the cortical column. Therefore if we can understand how a cortical column functions, we can understand the entire cortex [29].

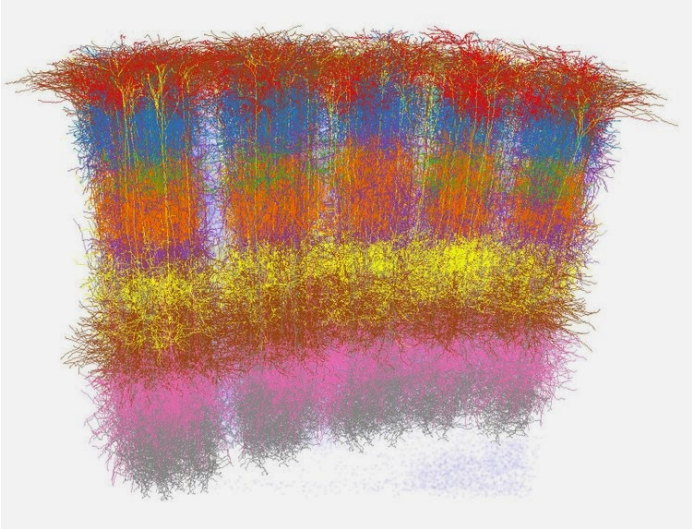


Figure 3.3: 3D reconstruction of five neighboring columns in rat vibrissal cortex[1]

3.2.1 Sparse Distributed Representations

A key to understanding how the brain functions is understanding how the brain perceives the world. Our senses need to be encoded into a common language that is compatible with the circuitry of the cortex, and understanding how that process works is an enormous challenge itself. As an example, consider the field of Natural Language Processing. It is a field that has had an enormous amount of work done for decades and yet NLP is still incapable of human-like interpretation of text. One

[1]Image credit: Marcel Oberlaender et al.

system of encoding that hopes to provide a language for cortical circuitry is sparse distributed representations.

Consider the binary representation of the integer 32 versus that of the integer 31. The two numbers are quite similar: they are only 1 apart, so they are as close as two integers can be. Yet their binary representations are [100000] and [011111] respectively. They have no shared "on" bits so despite their similarity, their binary representations reflect none at all. In fact, despite being as close as two integers can be (a Euclidean distance of 1), their binary representations have a Hamming distance, the number places in which two codewords differ, of 5, the maximum Hamming distance of two codewords of length 5 [1].

This means that our encoding does not preserve semantic similarity or a concept of distance between elements which is highly undesirable for an code because if there is some kind of error in the code we could end up decoding something meaning the opposite of what we were trying to convey. As an example, consider $\mathbb{Z}$ from 0 to 31 which is mapped to $GF(2)^6$ by their binary representation. The mapping of 31 is [11111] but a single error in transmission can easily lead to [01111] which would be decoded as 15. So a code Hamming distance (d_H) of one away ($\frac{1}{5}$ of the total metric) lead to an element 16 integers away ($\frac{1}{2}$ of the total metric). We would obviously like to avoid this so that errors in the transmission of our codes are either (1) correctable or (2) lead to a decoding that is as close as possible to our original element.

This is achievable by simply conveying more information in our code. In our binary representation any single error led to another valid codeword (a codeword which decoded to an element of the input/output set) which meant that no errors could be detected or corrected. By expanding our code length, we increase the number of codewords (multiplying by the cardinality of the code alphabet for each character

added) meaning that fewer errors will result in other valid codewords and can possibly be detected or corrected.

A key strategy with Sparse Distributed Representations (SDRs) is to encode semantic similarity, such as with our idea of distance in our motivating example. This helps us achieve our second goal because even if we increase the error-tolerance of our code, there is still some probability of an uncorrectable error and we would like that error to result in a codeword as close to the original codeword as possible. To give you a real world example, imagine an air traffic controller sending instructions to a aircraft and they need to tell it to turn down 7° to start its descent. Obviously, an error resulting in 8° or 6° being interpreted by the pilot are both preferable to 90°.

To achieve our goal, we employ sparse distributed representations. Just as with traditional dense binary representations, we will represent sparse distributed representations as vectors over their code alphabet, in this case GF(2). We call them **sparse** distributed representations because these vectors typically only have a small proportion of the components as 1. We will use Numenta's notation of letting $w_{\overrightarrow{x}}$ denote the number of components in an SDR $\overrightarrow{x}$ that are 1, so $w_{\overrightarrow{x}} = \|x\|_1$ [2].

Given our definition of distance, we could say that two decodings of sparse distributed representations, a and b, are equal if and only if the $d_H(a, b) = w_a = w_b$. This would mean that both vectors would have to have the same dimensionality, same number of on bits, and all on and off bits would have to match. This definition is good for "equals," but suppose we have a single error in transmission or a single component of our distributed system fails, equality would thus fail. In order to be able preserve the ability to subsample and thus to preserve fault tolerance, we therefore need a less stringent definition for decoding SDRs. Numenta refers to this as the *overlap*, which is

$$overlap(\overrightarrow{a}, \overrightarrow{b}) \equiv \overrightarrow{a} \cdot \overrightarrow{b} \equiv \sum_{0}^{n-1} a_i b_i \tag{3.2}$$

Thus, we say two SDRs $\overrightarrow{a}$ and $\overrightarrow{b}$ decode to the same element of the input space if and only if $overlap(a, b) \geq \theta$ where $\theta \leq w_a$ and $\theta \leq w_b$ [2]. Denote the function that determines if two sparse distributed representations decode to the same element of the input space using some θ and $overlap(\overrightarrow{a}, \overrightarrow{b})$, $match_\theta(\overrightarrow{a}, \overrightarrow{b})$ and it is a function from SDR×SDR $\longrightarrow$ {true, false}.

Given a set of sparse distributed representations with the same dimension, $X = \{\overrightarrow{x_1}, \overrightarrow{x_2}, ..., \overrightarrow{x_n}\}$, we can find the union of the vectors using the bitwise OR operation over the i^{th} position of the vectors in the set to produce the i^{th} position of $union(X)$ [2]. For example, given [0100] and [0010] the union would be [0110]. We say an SDR $\overrightarrow{y}$ is an element of the union of a set of SDRs, $\overrightarrow{X}$, if and only if $match_\theta(\overrightarrow{X}, \overrightarrow{y})$ [2].

3.2.2 Cortical Learning Algorithms

Hierarchical Temporal Memory aims to implement the functions of the neocortex as faithfully as possible, and unsurprisingly uses the same atomic unit of the cell. Modeled after the pyramidal neurons of the neocortex, these cells each take their own inputs and make their own predictions which the layer combines into a cohesive prediction [21]. They have three states: active from feed-forward input, active from lateral input (predictive state), and inactive. These first two states correspond to a short burst of action potentials in a neuron, and a slower steady burst of action potentials on a neuron respectively in our own brain.

Just like actual pyramidal neurons, HTM cells also dendrite segments. Each cell has one proximal dendrite segment and a dozen or more distal dendrite segments

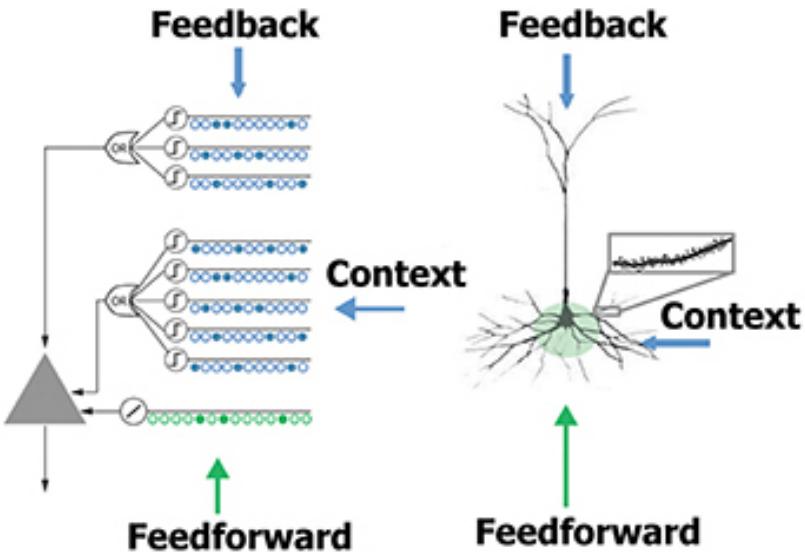


Figure 3.4: Pyramidal Neuron vs. Hierarchical Temporal Memory Cell [11]

[21]. Proximal dendrites accept feed-forward input and all cells in the same column respond to a similar feed-forward input due to a class of inhibitory cells. On the other hand, distal dendrites receive lateral input from nearby cells and their response is unique on a cell-by-cell basis dependent on the connected synapses on each cells distal dendrites [21]. Figure 3.4 illustrates the similarities between a pyramidal neuron and an HTM cell.

In a biological neuron synapses have real number weights with a stochastic nature, but in the HTM model synapses have binary weights (connected or disconnected) that are informed by real number permanence values [21]. These permanence values allow our artificial neurons to model forming and un-forming of "potential synapses" which represent axons that pass close enough to a dendrite segment that they could potentially form a synapse. Once the permanence is above a certain threshold (a parameter of the network) the synapse is considered connected.

Encoder

As discussed in Section 3.2.1, in the real world we need to encode sensory data to a format which the cortex can understand. The entire data pipeline of Hierarchical Temporal Memory can be seen in Figure 3.5. For numeric data, we generally use a Random Distributed Scalar Encoder. Just as with any scalar encoder, it is a function from a number to a binary vector with `width` 1 bits and the rest of the bits being zero. All scalar encoders also have either implicitly or explicitly a `resolution` parameter which determines the size of the "buckets" in which all numbers in that "bucket" are encoded to the same binary vector, this is essentially the lower-limit topology on $\mathbb{R}$. This is necessary because for a binary vector of length n there are only 2^n possibly options and infinite real numbers between any two numbers.

Figure 3.5: Data Pipeline of an HTM [11]

A region's input space and output space are the same; an HTM takes in a time series over the space and predicts an element or set of elements of the input space it believes to come next [31]. Each region has an input space which could be a single input set or the Cartesian product of multiple input sets. As an example, {0,1,2,3,4} could be an input set and if a parent region receives input from child region which has {0,1} as its input space and another child region with {red, green, blue} as its input space, then the parent region's input space would be {(0, red), (1, red), (0, green), (1, green), (0, blue), (1, blue)}. An encoding function from an input space to n-dimensional sparse distributed representations is not guaranteed to

be surjective or injective. We would not expect surjectivity, but could get it through nearest neighbor decoding, but injectivity is something that one would think would be preserved. However, because parent regions are allowed to take a slice of the output vector (not the "on" bits, just an arbitrary portion) when being fed to from multiple child regions, we cannot guarantee the property.

Spatial Pooler

The Spatial Pooler's job is to receive the encoded representation of the input space and convert the input to a sparse pattern—a requirement for learning and prediction in Hierarchical Temporal Memory theory [21]. Upon initialization, all of the potential synapses on the dendrite segments of all of the mini-columns are randomly assigned permanence values and based on these permanence values a potential synapse may be valid or not [21].

The pooler begins with an input from the encoder that has a fixed number of bits and for each column we determine how many valid synapses are connected to active input bits [21]. This number is then multiplied by a "boosting" factor determined dynamically by how often a column is active relative to its neighbors. The most highly activated mini-columns after boosting then disable a percentage of the mini-columns within a dynamically determined "inhibition radius" to create a sparse set of activated mini-columns [21]. Lastly, each active column updates the permanence values of its potential synapses. The permanence values associated with active input bits are increased while those associated with inactive bits are decremented [21].

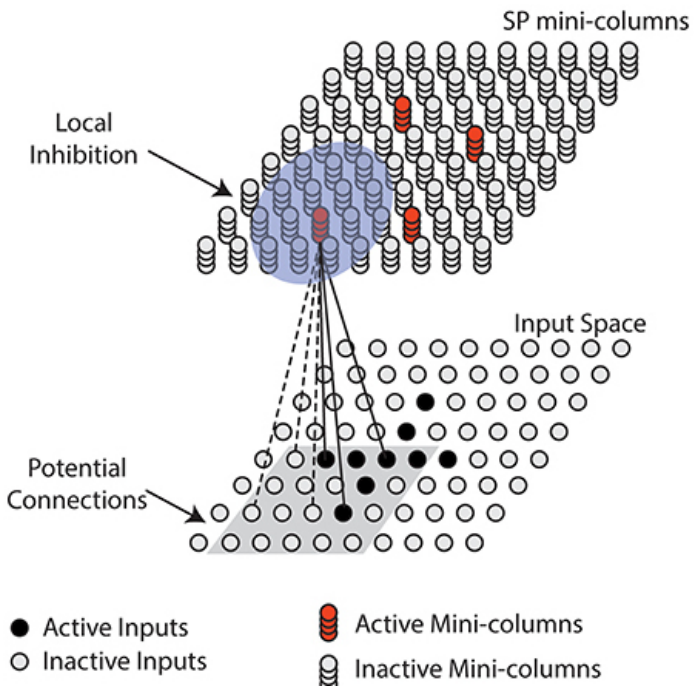


Figure 3.6: Spatial Pooler [11]

Temporal Pooler

The temporal pooler takes over where the spatial pooler left off, with a sparse set of activated mini-columns. For each active mini-column, if there are cells in the predictive states, activate them and in the absence of a cell in the predictive state, activate all of the cells in the mini-column [21]. The set of active cells is the HTM's representation of the current input in the context of the previous of the previous inputs [22]. Next, each cell's dendrite segment counts how many connected synapses are associated with active cells and if this number is above a certain threshold, the dendrite segment is activated. A cell with an active dendrite segment is put into a predictive state in the next time step [22]. The algorithm is illustrated in Figure 3.7.

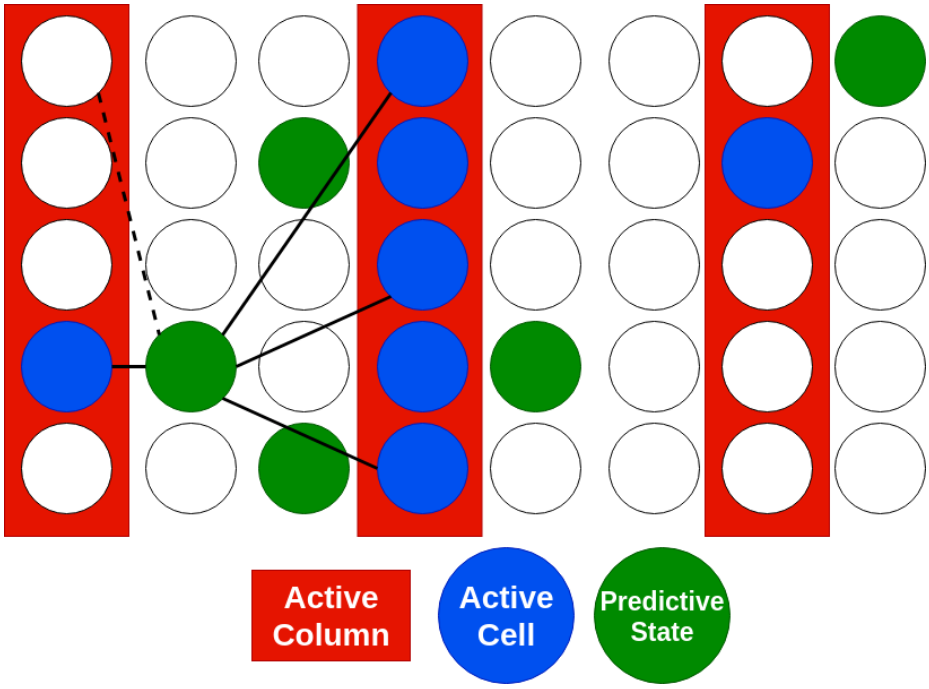


Figure 3.7: Temporal Pooler

Classifier

Once the system completes its processing of a time step's input, the system outputs a sparse distributed representation which may have gone through multiple layers and it is important to be able to decode the system's prediction in order for the system to be useful. Hierarchical Temporal Memory currently uses something called an SDR Classifier to decode the predictions of an HTM [13]. At its essence, an SDR Classifier is a single layer, feed forward, neural network [13].

SDR Classifiers are able to decode the information HTMs give about predictions n times steps in the future by maintaining a weight matrix [13]. The matrix's weights are adjusted to "learn" after each time step to reflect the correct weighting between the output vector at time t and the probability distribution of the input/output space

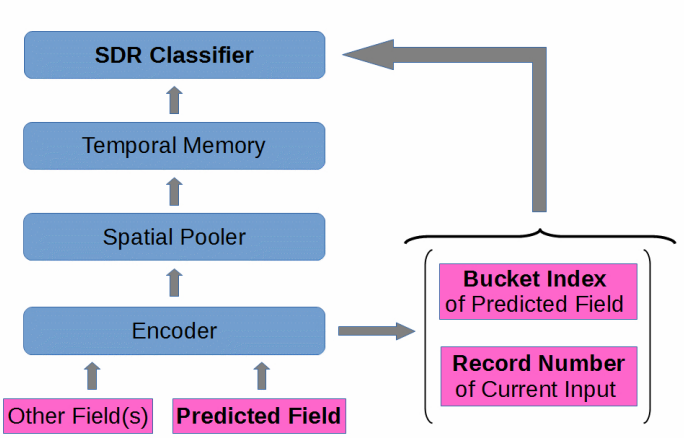


Figure 3.8: SDR Classifier

at time $t-n$ [13]. This enables the matrix to reflect relationships between inputs and outputs n time steps apart. To determine the SDR Classifier's interpretation of an output at time $t+n$, the SDR Classifier takes in the HTM's output vector and uses Softmax (Eqn 3.3) to determine the most likely decoding [13]. So for each of the k classes in the output space, the certainty that the output vector is referring to it is

$$y_j = \frac{e^{a_j}}{\sum_{i=1}^{k} e^{a_i}} \tag{3.3}$$

where a_j is the activation level of the j^{th} element of the output space, calculated by multiplying the output vector by the j^{th} column of the weight matrix element wise [13].

Chapter 4

Literature Review

There has been some work in evaluating Hierarchical Temporal Memory's ability to predict time series. Numenta themselves has published a paper, *Evaluating Real-Time Anomaly Detection Algorithms—the Numenta Anomaly Benchmark* in which they showed that the HTM algorithm outperformed other tested anomaly detectors for streaming data [25]. Their anomaly detection is also the center-piece of a number of applications on their HTM applications website[1] listing "rogue behavior" detection, geospatial tracking, and financial monitoring services. The IT analytics company Grok is another commerical application of HTM anomaly detection and is a partnership between Avik Partners and Numenta.

Outside of Numenta's work, there have been a few studies done on an HTM's ability to predict time series, two of which focused on financial time series. The first of these found "A fair amount of the generated models reached the goal of at least a 1.05 TP/FP ratio, and several models also made a profit" [4]. However, the author also noted, "there was more room for improvements through either larger

[1] https://numenta.com/machine-intelligence-technology/applications/

models or...through parameter optimization..." [4]. Gabrielsson built on this work and found that "results show that HTM models outperformed the neural network models," but possibly more interestingly found that "even though the number of bad trades (False Positives) outnumber the number of good trades (True Positives) in both the validation and test sets, all models still yielded positive PNLs [Profits Net Losses]" [15]. This could indicate that HTMs are not outperforming in terms of all prediction, simply anomaly prediction.

Two contrasting views are found in papers that more directly looked at the time series capabilities of Hierarchical Temporal Memory. The first, a Master's Thesis by Michael Galetzka, asked an HTM to predict the navigation behavior of visitors to a website using publicly available data from `msnbc.com` [16]. The paper found that the HTM had a "surprisingly high" prediction accuracy for the kind of noisy data it was given and claims that the results are comparable or better than state-of-the-art approaches, but acknowledges that comparisons are difficult due to a lack of a uniform performance metric across papers.

Most relevant to this exploration of ARMA models is Vivmond's *Utilizing the HTM algorithms for weather forecasting and anomaly detection.* This paper applied HTMs to weather time series such as wind direction, wind speed, wind pressure, precipitation, humidity, and temperature some of which are quite literally textbook examples of ARMA models [6, 35]. Vivmond found that "its prediction results were for the most part outperformed by a much simpler analogue method that used the average of the last few days as its forecast basis" [35].

In particular, he points to Hierarchical Temporal Memory's inability to generalize trends, saying "it is largely useless when seeing new values or when seeing previously seen patterns with different values" [35]. Vivmond also points to why he believes this

is the case stating, that if Hierarchical Temporal Memory "was to learn a pattern consisting of several numbers and then be given a series of new numbers arranged in a similar pattern to the previous one, but with either higher, lower, or stretched values (i.e. multiplied by common a factor), it would be unable to correctly predict any of them" [35].

Chapter 5

Experiment Design

5.1 Hardware and Software Specifications

We utilized the official Python (Python 2.7) implementation of Numenta's cortical learning algorithms, NuPIC[1] and its Network API[2]. At the time of my work NuPIC was at version 1.05 and we chose to use this version. It should be noted that although much of NuPIC is in Python, many of its computationally intensive algorithms are implemented in C++ using their NuPIC.core repository[3] and it also relies on a number of external libraries such as `numpy` and `pandas`.

Due to the computationally intensive nature of my research, we also heavily relied on parallel processing to run my code in a reasonable time frame. In particular, the `multiprocessing` package in Python was indispensable to my research, allowing me to speed my code up a factor of up to 28x when running independent tests or swarming.

[1] NuPIC repository: https://github.com/numenta/nupic

[2] Network API docs: http://nupic.docs.numenta.org/stable/api/network/

[3] NuPIC.core repository: https://github.com/numenta/nupic.core

Dijkstra (Asus Vivobook)	
CPU	Intel i5-8250U, 4-Core, 1.60GHz
RAM	24 GB
OS	Linux Mint 19 Cinnamon
Galois (HP DL380 G7)	
CPU	2x Intel Xeon X5650, 6-Core, 2.70 GHz
RAM	192 GB
OS	Linux Mint 19 Cinnamon
Aslan (HP DL380 G9)	
CPU	2x Intel Xeon E5-2690, 10-Core, 3.00 GHz
RAM	256 GB
OS	Gentoo

Table 5.1: Hardware Specifications

On the hardware side of things, the majority of development and testing was conducted on my personal laptop, an Asus Vivobook with some small upgrades. The code was then deployed to local servers which had more cores, more memory, and a higher clock speed allowing me to get faster results. For a synopsis of the hardware specifications, see Table 5.1.

5.2 Training Routine

Our proposed exploration of how well a Hierarchical Temporal Memory network is able to predict ARMA models requires that we are able to train an HTM network. These experiments required a flexible training routine that would allow us to train an

HTM on a variety of of different sequences which may require different parameters.

To do this, we drew on the training techniques of typical machine learning algorithms: we divided our data (in our case a sample from an ARMA model) into a training set, a validation set, and an evaluation set. The sequence was partitioned such that the training set was 80% of the instance and the validation and evaluation sets were both 10% of the sequence. To keep the training set continuous, we chose the validation and evaluation sets from the beginning or end of the initial sequence with equal probability and the order in which we assigned the either validation or evaluation was also chose with equal probability.

Before an HTM enters the classical training however, we added a phase before that in which the spatial pooler could encounter the data without the temporal pooler being turned on. We hoped this would allow the spatial pooler to form some of its synapses before turning on the temporal pooler and thus have the temporal pooler learn on less "noise.". The number of iterations was a parameter we tuned* called `sibt` (**S**patial pooler **I**terations **B**efore **T**emporal pooler).

Then the classical machine learning techniques kicked in with the HTM learning its test set, then being tested on its validation set until performance stagnated or worsened, at which point we tested the HTM on the evaluation set. Performance was either evaluated as binary meaning

$$\text{Error}_{binary}(x, y) = \begin{cases} 1, & \text{if } x \neq y \\ 0, & \text{else} \end{cases} \tag{5.1}$$

or root mean squared error (RMSE) which is defined as

*tuned through swarming

$$\text{Error}_{RMSE}(x, y) = \sqrt{(x - y)^2} \tag{5.2}$$

There are a few small differences in that we added a `iter_per_cycle` parameter* which allowed us to run on the test and validation set multiple times, summing the performances, before comparing to the previous error, which would be useful for dealing with noise in the performance metric. We also added a `max_cycles`* parameter that acted as a cut-off switch, stopping training after that many cycles. Most importantly, a weighting scheme as added (`weights`*) which allowed us to weight the predictions in any way from 1 to n steps. In particular, we swarmed on weights for two through nine steps, leaving one at a constant 1.0.

5.3 Time Series Generation and Identification

The experiments required for this work required tools for analyzing and generating ARMA models, and to accomplish this, we utilized the `statsmodels`[4] Python module. To generate ARMA models with the desired characteristic, we were able to utilize a function from the `arma_generate_sample` function which is capable of producing an n-term sample from an ARMA model given AR and MA lag polynomials and a standard deviation for the noise term [32]. To produce random stationary (p,q)-ARMA models, we generated a random number between one and a thousand for each coefficient of the lag polynomial and normalized the AR and MA polynomials separately such that the sum of the coefficients for AR and MA were each less than or equal to one.

[4]statsmodels webpage: https://www.statsmodels.org/dev/index.html

In an attempt to avoid instances of the same model requiring different parameters, and namely different Random Distributed Scalar Encoder resolution, the samples of the ARMA processes were normalized such that the values were bounded between [-100,100]. However, this scaling had a large effect on the standard deviation of the resultant series which still needed to be understood. To measure the standard deviation of the post-scaling series, we utilized the `ARMA.fit` method which takes a p and q alongside the time series and returns the (p, q)-ARMA model using exact maximum likehood via Kalman filter [32].

A key component of this work was identifying ARMA time series, which fell into two steps: time series classification and then coefficient fitting. The two-step procedure was a necessity because the order of the time series are inputs to coefficient fitting, thus we had to determine the order of the series first (classification) and then fit the coefficients of the lag polynomials given that classification.

To accomplish classification we again utilized the `statsmodels` module, specifically the `arma_order_select_ic` function which finds information criterion for ARMA models. We chose to use Bayes Information Criterion for all further experiments. As discussed in Section 2.4, the Bayes Information Criterion (BIC) provides the order (p and q) of the time series

With the order of the ARMA model now identified, we utilized the `ARMA.fit` method which takes a p and q alongside the time series and returns the (p, q)-ARMA model using exact maximum likehood via Kalman filter, allowing us to get the coefficients of the lag polynomials [32]. The likelihood model used by `ARMA.fit` has the option to use a variety of methods [32]. We wrote a function which initially attempted to use the default solving method of limited memory Broyden-Fletcher-Goldfarb-Shanno and upon throwing an exception would then sequentially fallback upon the other

methods available: Broyden-Fletcher-Goldfarb-Shanno, Newton-Rhapson, Nelder-Mead, conjugate gradient, non-conjugate gradient, and Powell, in that order [32]. The accuracy of this function and the `arma_order_select_ic` function would therefore need to be assessed before attempting to draw conclusions based on their outputs because of how central they were to this work.

Chapter 6

Software Validation

6.1 Accuracy of `statsmodels` Tools

For any of the following experiments to mean anything, it was crucial that we were certain we could analyze ARMA models with a high degree of accuracy as so much of the proposed work depended on it. Not only was the `statsmodels` module used to generate the instances of the ARMA models used, we also needed to depend on it to classify time series and fit the coefficients of the lag polynomials. Relating to the generation of time series, our first task was to attempt to understand the relationship between input sigma and the standard deviation of the white noise post-scaling.

6.1.1 Measuring White Noise of ARMA Models

Before determining how the scaling affected the standard deviation of the white noise, it was of course important to ensure we were using an accurate tool. The `ARMA.fit` method returns an `ARMAResults` instance which along with the lag coefficients also contains other information about the fitted ARMA model such as `sigma2`

which is the variance of the residuals [32]. Taking the square root of this value should then produce the standard deviation of the resulting ARMA model.

To test the accuracy of the standard deviation produced by this tool, we measured the standard deviations of 270 instances of a (4,0)-ARMA model described by $\tilde{z}_t = 0.8\tilde{z}_{t-4} + a_t$ with input standard deviations ranging from one to nine with equal frequency (i.e. 30 with $\sigma = 1$, 30 with $\sigma = 2$, etc.). The results can be seen below in Figure 6.1. The data shows a strong correlation between input sigma and measured sigma, which allows us to conclude that the tool is reasonably accurate for input sigmas between 1 and 9.

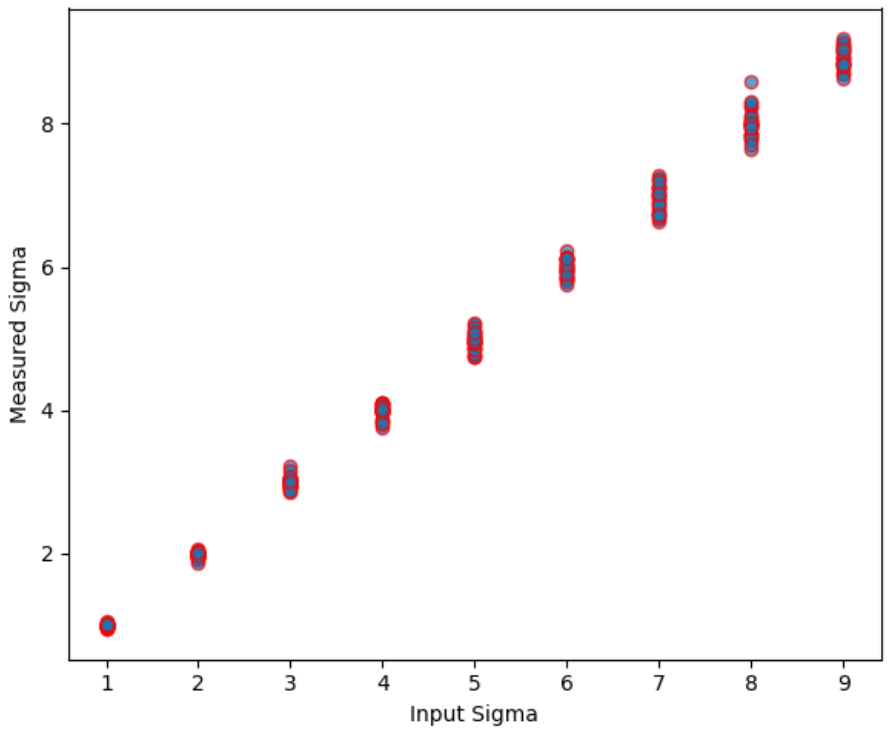


Figure 6.1: Standard Deviation of Generated Time Series Without Scaling

Using this tool, we then repeated the experiment, this time scaling the time series using the method described in Section 5.3 to better understand the affect scaling has on the standard deviation of the white noise. The results can be seen in Figure 6.2. In stark contrast to the previous plot, this data shows a lack of correlation and a much wider standard deviation for the distribution of measured (post-scaling) sigma. This suggests that a standard deviation between 14 and 22 was to be expected for all series regardless of the input sigma and for this reason we used an input sigma of 1 for all further experiments.

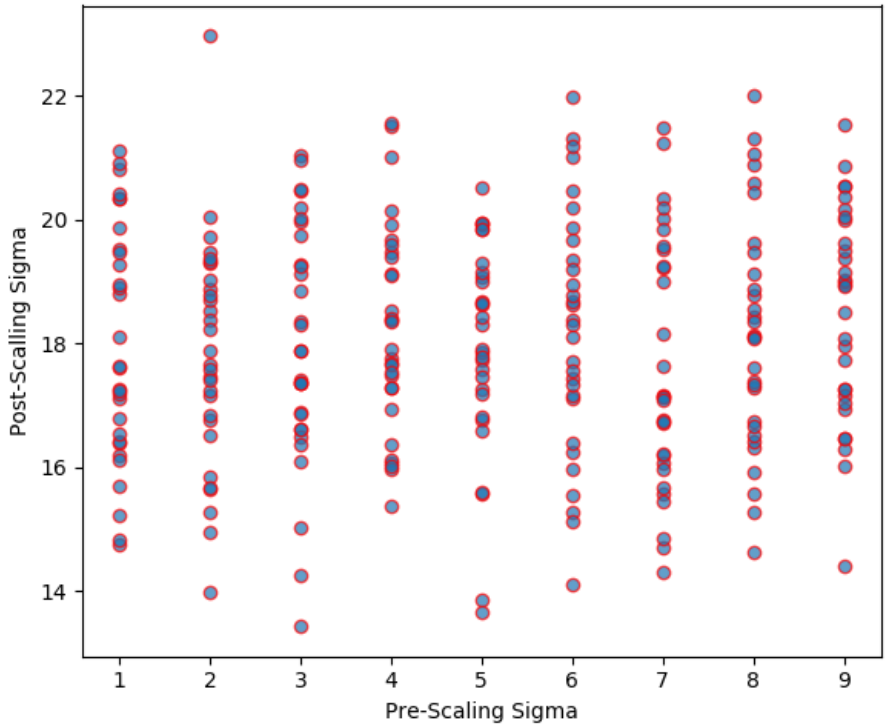


Figure 6.2: Standard Deviation of Generated Time Series Pre- and Post-Scaling

6.1.2 Time Series Classification

The next step was to assess the accuracy of the Bayes Information Criterion produced by the `arma_order_select_ic`. It was crucial that this information be accurate because the BIC produced by this function would be used as the order of the ARMA model used when fitting the coefficients of the lag polynomials.

To measure the accuracy of `arma_order_select_ic` we designed an experiment in which we would give the function 1500 instances of a known ARMA model so that we could compare the p and q produced by the function to the order of the model used to generate the instance. We chose to test both Autoregressive and Moving-Average models with orders ranging from three to six. For each experiment, we generated 1,500 instances of the given model and passed the instances to `arma_order_select_ic`.

Table 6.1 shows the results of the experiment for third-order Autoregressive and Moving-Average models with lag polynomial coefficients of [0.0, 0.2, 0.8]. The left-hand side shows the results for the Autoregressive model while the Moving-Average results are on the right. The occurrences column shows how many times that classification was returned by the tool, so for example if `arma_order_select_ic` classified an instance as a (4,2)-ARMA model, that would count towards an occurrence in the "AR Lag 4" cell and the "MA Lag 2" cell. The results show that the function is accurate, with over 99% of instances correctly classified for both Autoregressive and Moving-Average models.

Autoregressive Model			Moving-Average Model		
	Occurrences	Percent	Occurrences	Percent	
AR Lag 2	0	0.00%	0	0.00%	MA Lag 2
AR Lag 3	1492	99.47%	1490	99.33%	MA Lag 3
AR Lag 4	6	0.40%	9	0.60%	MA Lag 4
AR Lag 5	2	0.13%	1	0.07%	MA Lag 5
MA Lag 1	11	0.73%	7	0.47%	AR Lag 1
MA Lag 2	1	0.07%	2	0.13%	AR Lag 2

Table 6.1: BIC produced for (3,0)- and (0,3)-ARMA Models

The experiment was also conducted for fourth-order series, this time with lag coefficients of [0.0, 0.0, 0.0, 0.8]. The results are in Table 6.2, and again show that the tool is extremely accurate, with over 99% accuracy.

Autoregressive Model			Moving-Average Model		
	Occurrences	Percent	Occurrences	Percent	
AR Lag 3	0	0.00%	0	0.00%	MA Lag 3
AR Lag 4	1493	99.53%	1494	99.60%	MA Lag 4
AR Lag 5	4	0.27%	5	0.33%	MA Lag 5
AR Lag 6	3	0.20%	1	0.07%	MA Lag 6
MA Lag 1	10	0.67%	8	0.05%	AR Lag 1
MA Lag 2	2	0.13%	0	0.00%	AR Lag 2

Table 6.2: BIC produced for (4,0)- and (0,4)-ARMA Models

For fifth-order series, we assigned a lag polynomial of [0.0, 0.5, 0.0, 0.0, 0.5] for

both the Autoregressive and Moving-Average model. The results in Table 6.3 show that the tool is again highly accurate, but notably the Moving-Average accuracy in this case sunk slightly below 99% to 98.93%.

Autoregressive Model			Moving-Average Model		
	Occurrences	Percent	Occurrences	Percent	
AR Lag 4	0	0.00%	0	0.00%	MA Lag 4
AR Lag 5	1492	99.47%	1484	98.93%	MA Lag 5
AR Lag 6	8	0.53%	15	1.00%	MA Lag 6
AR Lag 7	0	0.00%	1	0.07%	MA Lag 7
MA Lag 1	5	0.33%	11	0.73%	AR Lag 1
MA Lag 2	0	0.00%	1	0.07%	AR Lag 2

Table 6.3: BIC produced for (5,0)- and (0,5)-ARMA Models

Lastly, the data in Table 6.4 shows results for both (6,0)-ARMA model and (0,6)-ARMA models with coefficients for the respective lag polynomials of [0.0, 0.0, 0.4, 0.0, 0.3, 0.3]. As shown in the table, 1476 (98.40%) and 1480 (98.67%) were classified correctly as sixth-order respectively, representing a slight drop in accuracy, but still very accurate.

The results in Tables 6.1, 6.2, 6.3, and 6.4 show that that the `statsmodels` tool for selecting Bayes Information Criterion is extremely accurate for these two models. It should be noted that Table 6.4 only shows $p \pm 2$ and $q \pm 2$ for a (p, q)-ARMA model because there were no occurrences outside of that range in this experiment.

Autoregressive Model			Moving-Average Model		
	Occurrences	Percent	Occurrences	Percent	
AR Lag 5	6	0.40%	4	0.27%	MA Lag 5
AR Lag 6	1482	98.80%	1483	98.87%	MA Lag 6
AR Lag 7	11	0.73%	11	0.73%	MA Lag 7
AR Lag 8	1	0.07%	2	0.13%	MA Lag 8
MA Lag 1	12	0.80%	5	0.33%	AR Lag 1
MA Lag 2	4	0.27%	4	0.27%	AR Lag 2

Table 6.4: BIC produced for (6,0)- and (0,6)-ARMA Models

6.1.3 Fitting Coefficients of Time Series

Having quantified the accuracy of the information criterion tool, it was necessary to assess the accuracy of the `fit` tool so that we could analyze the differences between the input model and the output sequence coefficient-by-coefficient. This would give us a statistical confidence associated with our result using the standard deviation of the differences between the model coefficient and the tool's output. We again used 1,500 instances, orders three through six (using the same lag polynomials as the last set of experiments), and both Autoregressive and Moving-Average models. Models past sixth-order were not tested because the time to fit coefficients using `fit` grew very quickly with respect to the order.

A summary of the results of this experiment can be seem in Tables 6.5, 6.6, 6.7, and 6.8. For each table, the results for instances of the Autoregressive model are on the left while the results for the Moving-Average model are on the right. μ_{diff} represents the mean difference between the model's coefficient and the coefficient produced by `fit`,

and the standard deviation is found under σ_{diff}. For all models and all coefficients, the observed mean difference for all coefficients is within a standard deviation of 0, indicating that the function is producing accurate coefficients. Histograms for the coefficients from the sixth-order experiment (Table 6.8) can be found in Appendix A. From the histograms, the differences appear normally distributed. Using this data, we can then measure a time series and determine how far from the mean it is in terms of standard deviations.

Autoregressive Model			Moving-Average Model		
	μ_{diff}	σ_{diff}	μ_{diff}	σ_{diff}	
AR Lag 1	0.0020	0.03149	0.00162	0.05368	MA Lag 1
AR Lag 2	-0.00088	0.03129	-0.00127	0.03097	MA Lag 2
AR Lag 3	0.00596	0.02015	-0.00151	-0.02120	MA Lag 3
AR Lag 4	0.00031	0.01852	0.00002	0.03803	MA Lag 4
AR Lag 5	-0.00047	0.02124	-0.00056	-.0215	MA Lag 5
MA Lag 1	0.00074	0.02678	0.00052	0.04943	AR Lag 1
MA Lag 2	-0.00066	0.02551	-0.00058	0.02553	AR Lag 2

Table 6.5: Errors on fitting coefficients to instances of (3,0)- and (0,3)-ARMA model

Autoregressive Model			Moving-Average Model		
	μ_{diff}	σ_{diff}	μ_{diff}	σ_{diff}	
AR Lag 1	-0.00118	0.04119	0.00010	0.03064	MA Lag 1
AR Lag 2	-0.00209	0.03995	0.00047	0.01930	MA Lag 2
AR Lag 3	-0.00075	0.01945	0.00051	0.02011	MA Lag 3
AR Lag 4	0.00458	0.01916	-0.00411	0.01960	MA Lag 4
AR Lag 5	-0.00013	0.02919	0.00025	0.01785	MA Lag 5
AR Lag 6	-0.00097	0.02862	0.00008	0.00295	MA Lag 6
MA Lag 1	-0.00057	0.03464	0.00053	0.02469	AR Lag 1
MA Lag 2	0.00047	0.01930	0.00000	0.00000	AR Lag 2

Table 6.6: Errors on fitting coefficients to instances of (4,0)- and (0,4)-ARMA model

Autoregressive Model			Moving-Average Model		
	μ_{diff}	σ_{diff}	μ_{diff}	σ_{diff}	
AR Lag 1	0.00062	0.0293	-0.00053	0.04604	MA Lag 1
AR Lag 2	0.00120	0.02771	-0.00157	0.02918	MA Lag 2
AR Lag 3	-0.00015	0.03318	0.00116	0.03725	MA Lag 3
AR Lag 4	-0.00184	0.02794	0.00036	0.02942	MA Lag 4
AR Lag 5	0.00335	0.02785	-0.00171	0.02857	MA Lag 5
AR Lag 6	0.00011	0.00696	-0.00004	0.02024	MA Lag 6
AR Lag 7	0.00000	0.00000	0.00000*	0.00000*	MA Lag 7
MA Lag 1	0.00037	0.01111	0.00000	0.03680	AR Lag 1
MA Lag 2	0.00000	0.00000	0.00016	0.00608	AR Lag 2

Table 6.7: Errors on fitting coefficients to instances of (5,0)- and (0,5)-ARMA model

Autoregressive Model			**Moving-Average Model**		
	μ_{diff}	σ_{diff}	μ_{diff}	σ_{diff}	
AR Lag 1	0.00349	0.07462	0.00276	0.05784	MA Lag 1
AR Lag 2	-0.00185	0.03404	-0.00147	0.04248	MA Lag 2
AR Lag 3	0.00134	0.02895	-0.00049	0.02883	MA Lag 3
AR Lag 4	-0.00039	0.03909	0.00192	0.03478	MA Lag 4
AR Lag 5	0.00093	0.02925	-0.00128	0.03096	MA Lag 5
AR Lag 6	0.00387	0.03895	-0.00175	0.03532	MA Lag 6
AR Lag 7	0.00042	0.01602	-0.00016	0.01331	MA Lag 7
AR Lag 8	0.00008	0.00314	-0.00010	0.00788	MA Lag 8
MA Lag 1	-0.00461	0.06797	-0.00193	0.04801	AR Lag 1
MA Lag 2	0.00085	0.01665	0.00142	0.03004	AR Lag 2

Table 6.8: Errors on fitting coefficients to instances of (6,0)- and (0,6)-ARMA model

6.2 Swarming for Parameter Optimization

As noted in Section 5.2, to find optimal or near optimal parameters for predicting ARMA models. We decided to swarm on two different sets of parameters, one set ("lite") constituting just RDSE resolution and training parameters and another which constituted a wider array of HTM parameters ("full"). We chose bounds on the parameters by looking at a variety of settings used in similar papers and the bounds for all parameters can be seen in Table 6.9.

	Param Type	**Lite or Full**	**min**	**max**
RDSE Resolution	HTM	Lite	0.5	10
SIBT	Training	Lite	0	50
n**Weight**	Training	Lite	0	10
IterPerCycle	Training	Full	1	3
potentialPct	HTM	Full	.00001	1
numActiveColumnsPerInhArea	HTM	Full	20	80
synPermConnected	HTM	Full	.00001	0.5
synPermInactiveDec	HTM	Full	.00001	.1
activationThreshold	HTM	Full	8	40
newSynapseCount	HTM	Full	15	35

Table 6.9: Parameters Swarmed On

In order to test the particle swarming algorithm, we first wanted to first give it problems where we knew the global minimum. We chose to use $f(x) = \frac{x_1^2}{2} + \frac{x_2^2}{4}$, an elliptic paraboloid with minimum of 0 at (0,0). Its graph can be seen in Figure 6.3. The particle swarm optimization function's best position and value at the position

for each iteration can be seen in Table 6.10 where the minimum is found.

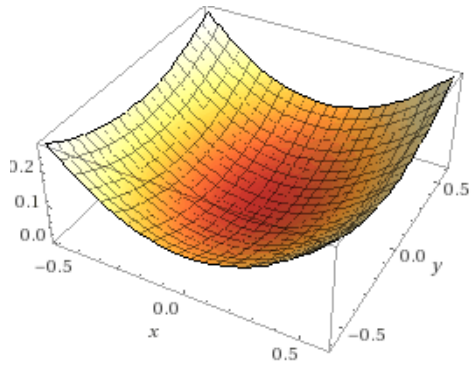


Figure 6.3: Elliptic Paraboloid

Iteration	**Value**	x_1	x_2
1	1430.59979	47.2942428328025	35.3398974845823
2	27.53131	-6.5412659780099	-4.95468729013965
3	27.38731	-0.017743951570481	10.4665480436202
4	27.38731	-0.017743951570481	10.4665480436202
5	27.38731	-0.017743951570481	10.4665480436202
6	27.38731	-0.017743951570481	10.4665480436202
7	27.38731	-0.017743951570481	10.4665480436202
8	27.38731	-0.017743951570481	10.4665480436202
9	27.38731	-0.017743951570481	10.4665480436202
10	27.38731	-0.017743951570481	10.4665480436202
11	27.38731	-0.017743951570481	10.4665480436202
12	27.38731	-0.017743951570481	10.4665480436202
13	27.38731	-0.017743951570481	10.4665480436202
14	11.94353	4.31367687248319	3.24938756386777

Iteration	**Value**	x_1	x_2
15	11.94353	4.31367687248319	3.24938756386777
16	0.77440	1.24262366689631	-0.096838341728018
17	0.77440	1.24262366689631	-0.096838341728018
18	0.77440	1.24262366689631	-0.096838341728018
19	0.77440	1.24262366689631	-0.096838341728018
20	0.77440	1.24262366689631	-0.096838341728018
21	0.77440	1.24262366689631	-0.096838341728018
22	0.32076	0.596928275580422	-0.755249816475236
23	0.32076	0.596928275580422	-0.755249816475236
24	0.01062	-0.033690147634675	-0.200485668701763
25	0.01062	-0.033690147634675	-0.200485668701763
26	0.01062	-0.033690147634675	-0.200485668701763
27	0.01062	-0.033690147634675	-0.200485668701763
28	0.01062	-0.033690147634675	-0.200485668701763
29	0.01062	-0.033690147634675	-0.200485668701763
30	0.00101	0.028590224603503	-0.049107936470873
31	0.00101	0.028590224603503	-0.049107936470873
32	0.00007	0.008126388767238	-0.01152860694564
33	0.00007	0.008126388767238	-0.01152860694564
34	0.00005	-0.008683959756985	0.007775437476341
35	0.00005	-0.005826573477984	0.011779521851548
36	0.00001	-0.004689426574613	0.00016966603041

Table 6.10: Particle Swarming on an Elliptic Paraboloid with 24 Particles

The elliptic paraboloid is a rather trivial surface to optimize because the only minima is the global minimum. Therefore, we needed to test the algorithm on a surface which contained more local minima. Drawing from a collection of test functions for optimization[1], we then turned to the Shubert Function. The function can be see in Figure 6.4 and is defined as

$$f(x) = \left(\sum_{i=1}^{5} i\cos((i+1)x_1 + i)\right)\left(\sum_{i=1}^{5} i\cos((i+1)x_2 + i)\right)$$

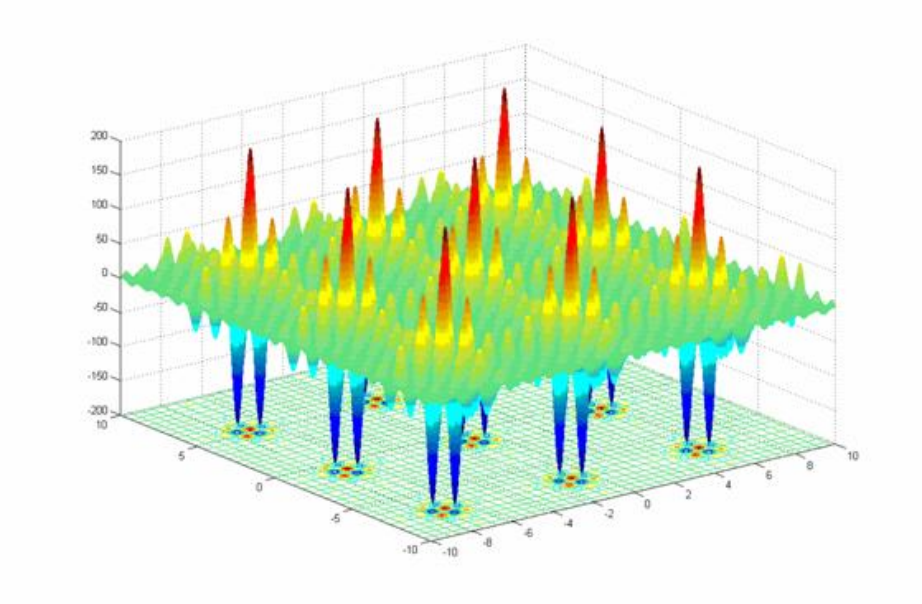


Figure 6.4: Shubert Function

Table 6.11 shows the algorithm's best value and position over 36 iterations, using 24 particles. It shows that the algorithm gets extremely close to one of the eighteen global minima, the value of which are -186.7309.

[1]Visual Library of Simulation Experiments: http://www.sfu.ca/ ssurjano/optimization.html

Iteration	**Value**	x_1	x_2
1	-27.7871597266311	-1.56156360629459	-9.40405561123859
2	-38.9415387245841	-1.59702841825173	-6.81484370344231
3	-38.9415387245841	-1.59702841825173	-6.81484370344231
4	-90.4499474261868	-0.874002681907612	-6.35272911905053
5	-90.4499474261868	-0.874002681907612	-6.35272911905053
6	-90.4499474261868	-0.874002681907612	-6.35272911905053
7	-90.4499474261868	-0.874002681907612	-6.35272911905053
8	-158.871990679088	5.52843458731665	-7.81257030731857
9	-158.871990679088	5.52843458731665	-7.81257030731857
10	-158.871990679088	5.52843458731665	-7.81257030731857
11	-158.871990679088	5.52843458731665	-7.81257030731857
12	-158.871990679088	5.52843458731665	-7.81257030731857
13	-158.871990679088	5.52843458731665	-7.81257030731857
14	-158.871990679088	5.52843458731665	-7.81257030731857
15	-158.871990679088	5.52843458731665	-7.81257030731857
16	-158.871990679088	5.52843458731665	-7.81257030731857
17	-158.871990679088	5.52843458731665	-7.81257030731857
18	-158.871990679088	5.52843458731665	-7.81257030731857
19	-158.871990679088	5.52843458731665	-7.81257030731857
20	-158.871990679088	5.52843458731665	-7.81257030731857
21	-158.871990679088	5.52843458731665	-7.81257030731857
22	-158.871990679088	5.52843458731665	-7.81257030731857
23	-184.93002874917	5.49361129961916	-7.68261917965731

Iteration	**Value**	x_1	x_2
24	-184.93002874917	5.49361129961916	-7.68261917965731
25	-185.961916264334	5.48292563466807	-7.69019184033006
26	-185.961916264334	5.48292563466807	-7.69019184033006
27	-185.961916264334	5.48292563466807	-7.69019184033006
28	-186.382912186899	5.47062619099101	-7.71098357453216
29	-186.712945809853	5.48176711558784	-7.71087715054839
30	-186.712945809853	5.48176711558784	-7.71087715054839
31	-186.712945809853	5.48176711558784	-7.71087715054839
32	-186.716820099857	5.48239393478027	-7.71073025043748
33	-186.728738465664	5.48311224966539	-7.70924814956375
34	-186.728860930707	5.48367397023694	-7.70882075560373
35	-186.730775369541	5.48310881675332	-7.70829063914814
36	-186.73077537	5.48310881675332	-7.70829063914814

Table 6.11: Particle Swarming on the Shubert Function with 24 Particles

Lastly, we wanted to test the swarming algorithm on a function with steep drop-offs to see how it interacted with sudden changes in the value of the function. To do this, we opted to use the Michalewicz Function which can be seen in Figure 6.5. The global minimum of the Michalewicz Function of two-dimensions is -1.8013. This minimum was found to four decimal places by iteration 33 with 24 particles, as shown in Table 6.12.

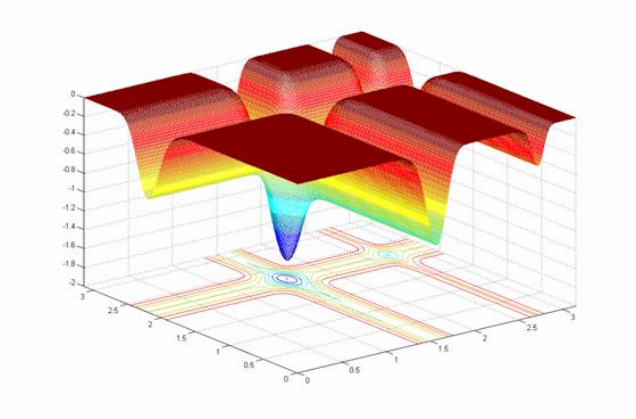


Figure 6.5: Michalewicz Function

Iteration	**Value**	x_1	x_2
1	-0.799480456112465	2.21347770457907	0.143961547047909
2	-1.02001519034968	2.14360380905455	1.38237622407863
3	-1.70300638818231	2.20239464178804	1.62052034936591
4	-1.70300638818231	2.20239464178804	1.62052034936591
5	-1.70300638818231	2.20239464178804	1.62052034936591
6	-1.70300638818231	2.20239464178804	1.62052034936591
7	-1.70300638818231	2.20239464178804	1.62052034936591
8	-1.70300638818231	2.20239464178804	1.62052034936591
9	-1.70300638818231	2.20239464178804	1.62052034936591
10	-1.70300638818231	2.20239464178804	1.62052034936591
11	-1.70300638818231	2.20239464178804	1.62052034936591
12	-1.70300638818231	2.20239464178804	1.62052034936591
13	-1.70300638818231	2.20239464178804	1.62052034936591
14	-1.70300638818231	2.20239464178804	1.62052034936591

Iteration	**Value**	x_1	x_2
15	-1.70300638818231	2.20239464178804	1.62052034936591
16	-1.70300638818231	2.20239464178804	1.62052034936591
17	-1.70300638818231	2.20239464178804	1.62052034936591
18	-1.71565574491555	2.2744913395185	1.58078343903939
19	-1.79362071947288	2.1990117512713	1.58431410110791
20	-1.79362071947288	2.1990117512713	1.58431410110791
21	-1.79362071947288	2.1990117512713	1.58431410110791
22	-1.79362071947288	2.1990117512713	1.58431410110791
23	-1.79826958592552	2.18944239110428	1.56914498663819
24	-1.80069249796879	2.20426325930944	1.57457971159592
25	-1.80069249796879	2.20426325930944	1.57457971159592
26	-1.80112250999064	2.20615857272585	1.57033327455812
27	-1.80112250999064	2.20615857272585	1.57033327455812
28	-1.80112250999064	2.20615857272585	1.57033327455812
29	-1.80116975572391	2.20378201146668	1.57252508988759
30	-1.80120553201305	2.20287445946326	1.57235006778859
31	-1.80128407656523	2.20262592357202	1.57012834876345
32	-1.80128706813547	2.20229068042156	1.57029440752734
33	-1.80130139494625	2.203221148287	1.57089518231883
34	-1.8013027087429	2.20276797349352	1.57089494204502
35	-1.8013027087429	2.20276797349352	1.57089494204502
36	-1.80130328878	2.20293342486416	1.57074452816047

Table 6.12: Particle Swarming on the Michalewicz Function with 24 Particles

6.3 Training HTMs on Time Series

Our first challenge was to feed in-memory time-series to a Hierarchical Temporal Memory network. At the time of writing, We was not able to find any way to do this in NuPIC so we needed to develop a stream, which we titled *TimeSeriesStream*, to get our data to the HTM. Then we needed to know that we could train an HTM.

Writing test cases would be extremely difficult because of the difficult-to-predict ways in which HTMs are initialized, form synapses, and use inhibition to produce sparse representations. Additionally, NuPIC's code is spread across Python and C++ further complicating the process. Therefore, to monitor the internal workings of the HTM and check additional code written for training and calculating errors were all working correctly, we employed logging and CSV outputs of HTM predictions. This combination would allow us to track an HTM's learning process and see how an HTM's predictions over various time steps compared to the input sequence.

Due to the stochastic nature of ARMA models, we elected to begin with simple, deterministic series. Our first series of tests used sequences which did not require any temporal context past the last term-that is sequences such that you can know for certain the next term in the sequence by only knowing the last term. Examples include 1,2,1,2,...; 1,3,1,3,..; and 78,79,78,79,... . This allowed us to ensure that we could train and predict using only an HTM's spatial pooler.

Next, we trained Hierarchical Temporal Memory networks using deterministic sequences which relied on some more long-range temporal dependencies. For these patterns, it is not always sufficient to know just the last term, but you may need to also know some term other previous term which would require the HTM's temporal pooler. Examples include 1,2,3,2,.. and 1,2,2,3,5,2,2,4,1,...

Once we were confident in our ability to train HTMs on simple sequence, our next

task was to find the correct parameters for the ARMA models we wanted to train on. However, when using swarming to find optimal or near-optimal settings for ARMA models, we found that the variation in the instances caused by different innovation terms added noise to the performance metric, meaning optimal parameters would be extremely difficult to find, but using parameters produced by the swarming would still allow us to use better parameters than simply guessing.

We also needed to acquire a baseline for what level of error would be considered "accurate" for these scaled ARMA models. To do this, we used a single instance of fourth-order Autoregressive model, and fitted the coefficients using both `arma_order_select_ic` and `fit`, then compared the root mean squared error between the two sequences. The results from this experiment can be seen in Figure 6.6. We see that the mean RMSE seems to be in the high teens and bounded between 13 and 22%.

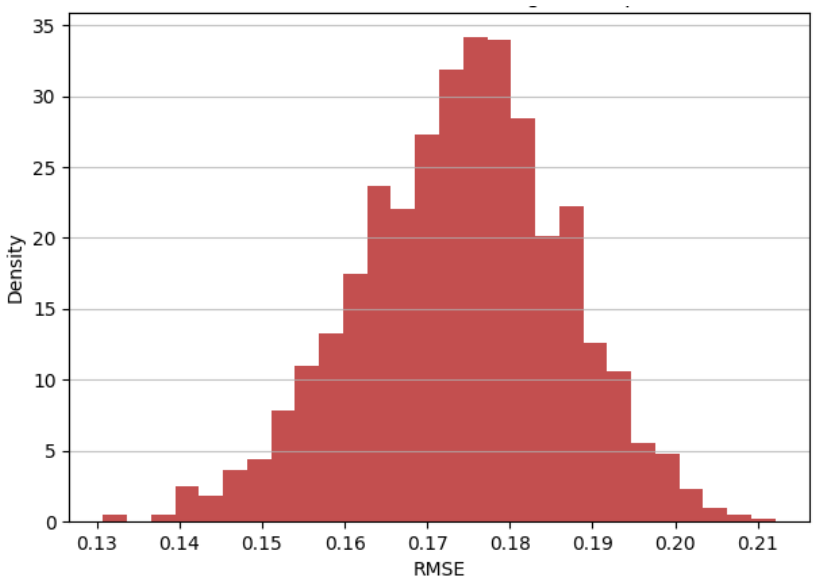


Figure 6.6: RMSE between an AR Instance and Instances from a Fitted Model

Chapter 7

Results

The central question of this research was: if a Hierarchical Temporal Memory network is given an ARMA time series, how does its predicted time series compare? Do accomplish this, we needed to train Hierarchical Temporal Memory networks on ARMA time series, classify and identify the appropriate ARMA models for their outputs, and determine with some measure of statistical certainty if the ARMA model which best describes the HTM predictions is the same ARMA model that produced the input. We also wanted to test the ability for an HTM to generalize learned patterns, which was called into question by papers discussed in Section 4.

7.1 Evaluating HTM's Ability to Generalize

A question that we wanted to explore was the one posed by Vivimond (see Section 4) where he postulated about Hierarchical Temporal Memory's ability to generalize, stating "was to learn a pattern consisting of several numbers and then be given a series of new numbers arranged in a similar pattern to the previous one, but

with either higher, lower, or stretched values (i.e. multiplied by common a factor), it would be unable to correctly predict any of them." We tested this theory by training an HTM network on an AR time series with coefficients [0, 0, -0.4, 0, -0.3, -0.3], assessing its ability to predict, then scaling the series, and asked the HTM to predict the series twice, once with learning turned off and then again with learning on. This was repeated one-hundred times.

For the first experiment, nothing was changed about the HTM network between pre- and post-scaling. The results are summarized in Table 7.1 which show that without learning enabled, a Hierarchical Temporal Memory network is very inaccurate at predicting a series after the series it trained on has been scaled. This is illustrated in Figure 7.3 which shows a trial of the experiment where Figure 7.2 is the original sequence with HTM predictions. However, Table 7.1 also shows that the RMSE of predictions decreased back to within a standard deviation of pre-scaling RMSE, indicating that the learning mechanism is resilient to scaling. This is illustrated in Figure 7.4. The plots in Figure 7.1 there is a positive correlation between Post-Scaling, Learning RMSE and Scaling Factor, meaning that as the scaling factor rises, the root mean squared error of the post-scaling predictions with learning on also tend to rise.

	Pre-Scaling	**Scaling Factor**	**Scaled, No Learning**	**Scaled, Learning**
μ	9.676%	10.560	24.873%	10.591%
σ	1.761%	3.225	9.659%	2.043%

Table 7.1: RMSE of HTM Predictions Before and After Scaling the Sequence

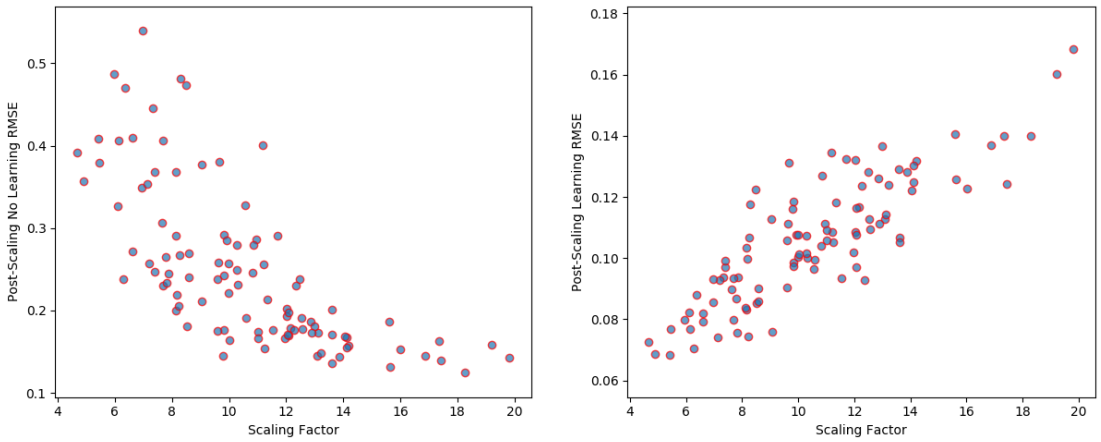


Figure 7.1: Scatterplots of Scaling Factor vs. Post-Scaling RMSE with and without Learning

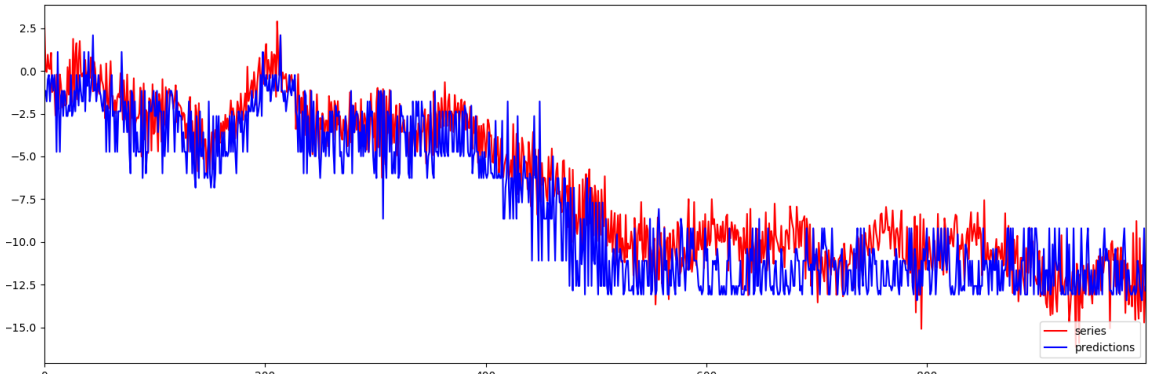


Figure 7.2: Pre-Scaling Sequence with HTM Predictions After Training

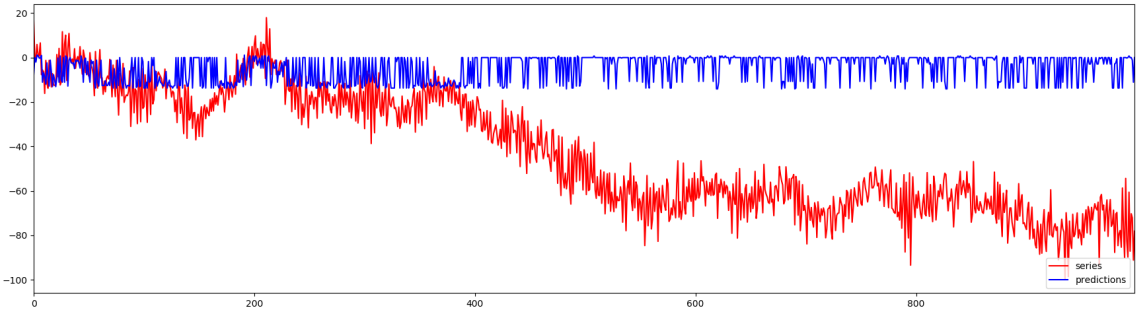


Figure 7.3: Post-Scaling Sequence with HTM Predictions Without Learning

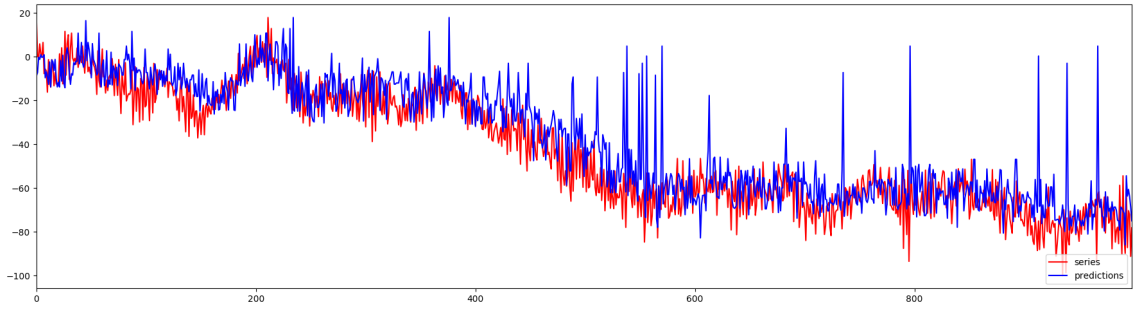


Figure 7.4: Post-Scaling Sequence with HTM Predictions With Learning

For our second experiment, we wanted to see if scaling the Random Distributed Scalar Encoder by the same factor as the sequence would lead to better post-scaling, no learning RMSE. Thus the only change to the HTM between the pre-scaling training and post-scaling assessment was to change the RDSE resolution by the same factor that we scaled the series by. One instance of the pre-scaling sequence with predictions can be seen in Figure 7.6 and the associated post-scaling sequence with predictions can be seen in Figure 7.7. Upon turning on learning again, we found that the RMSE again improved, which can be seen in Figure 7.8. A positive correlation between Post-Scaling, Learning RMSE and Scaling Factor, is again evident in the data, as illustrated in Figure 7.5.

	Pre-Scaling	**Scaling Factor**	**Scaled, No Learning**	**Scaled, Learning**
μ	9.604%	10.198	27.232%	10.512%
σ	2.094%	3.431	10.227%	2.430%

Table 7.2: RMSE of HTM Predictions Before and After Scaling the RMSE and Sequence

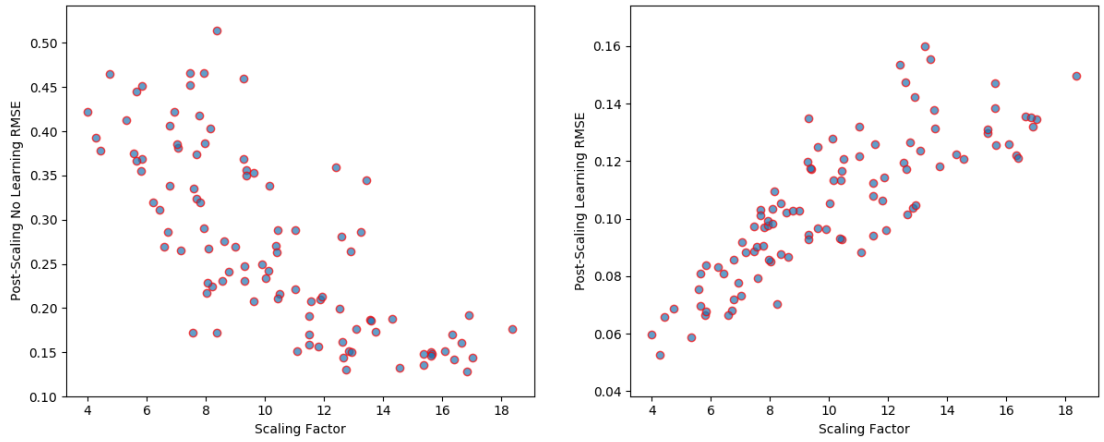


Figure 7.5: Scatterplots of Scaling Factor vs. Post-Scaling RMSE with and without Learning with Scaled RDSE Resolution

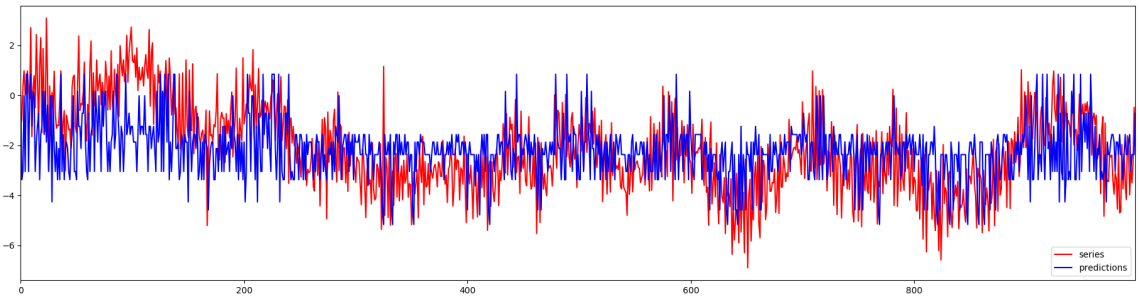


Figure 7.6: Pre-Scaling Sequence with HTM Predictions After Training with Scaled RDSE Resolution

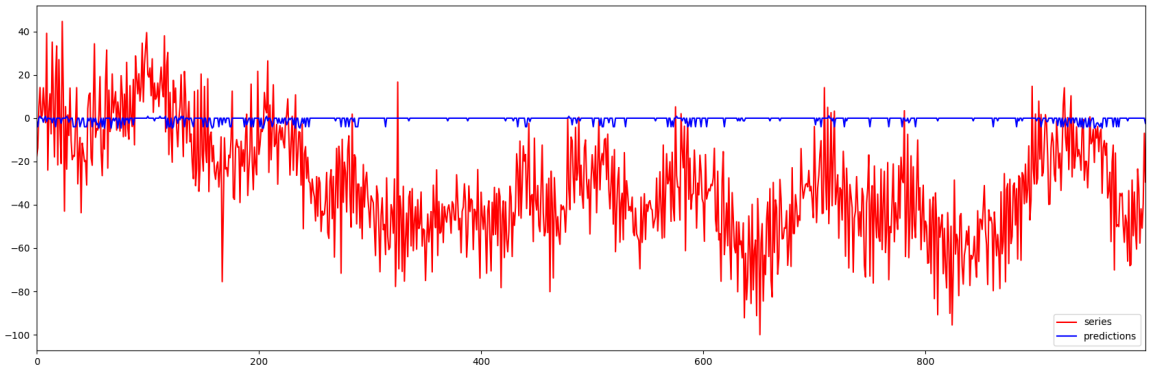


Figure 7.7: Post-Scaling Sequence with HTM Predictions Without Learning with Scaled RDSE Resolution

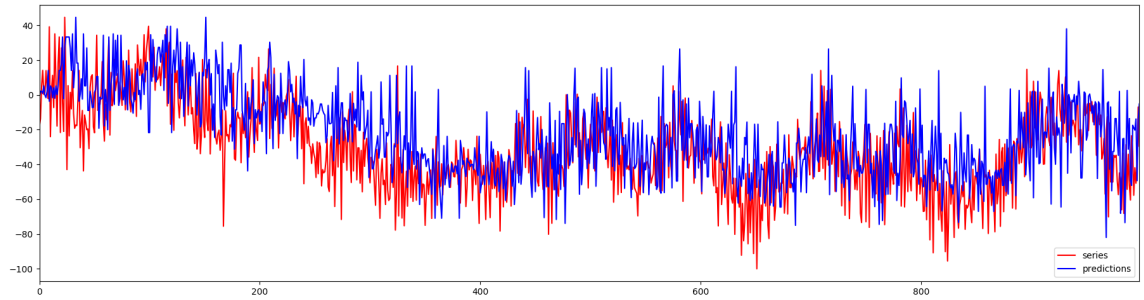


Figure 7.8: Post-Scaling Sequence with HTM Predictions With Learning with Scaled RDSE Resolution

7.2 Evaulating HTM Predictions

Our experiment required that we train a Hierarchical Temporal Memory network on sequences produced by ARMA models, fit its output, and assess with some statistical confidence it is the same model that we input. For our first trial, we decided to swarm for parameters and then train an HTM on the same instance of an AR model (done through seeding the random number generator). Examples of the sequences and predictions used in this section can be see in Figures 7.9 and 7.10.

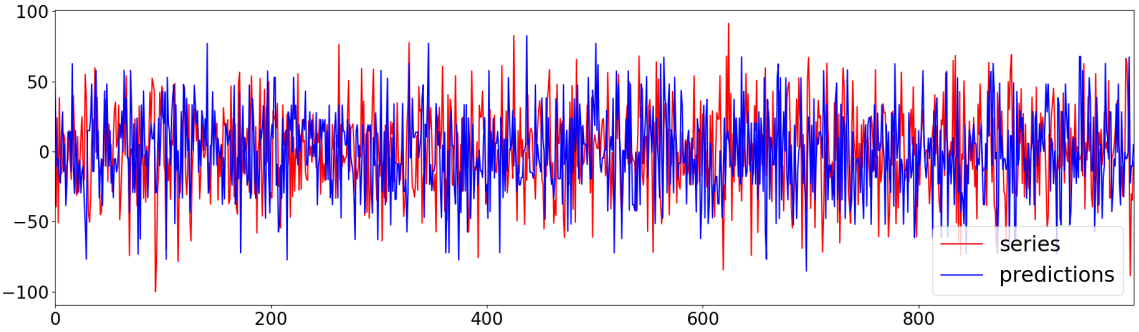


Figure 7.9: HTM One-Step Predictions of a (6,0)-ARMA Model

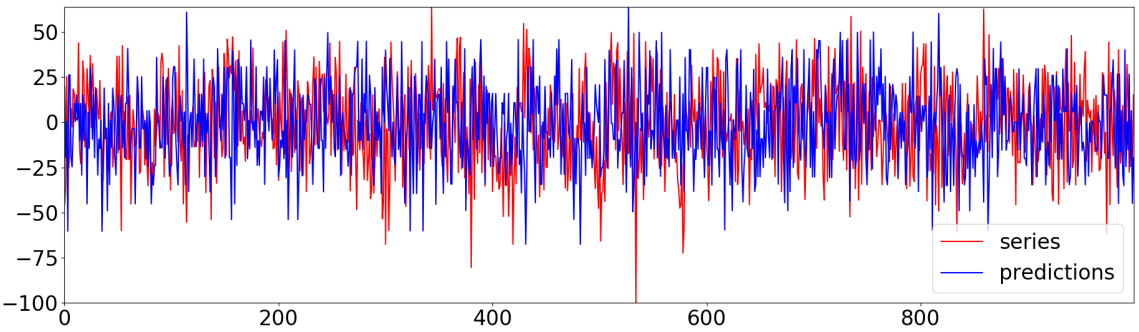


Figure 7.10: HTM One-Step Predictions of a (0,6)-ARMA Model

We fed Hierarchical Temporal Memory networks the two aforementioned AR and

MA models, gave their outputs to the `arma_order_select_ic` function, then the `fit` function, and finally subtracted the observed means, divided by the standard deviation, and took the absolute value to get the observations distance from the mean in terms of standard deviations. This was repeated using the lite and full parameters we found through swarming.

Autoregressive Model			Moving-Average Model		
	HTM Output	Z-Score	HTM Output	Z-Score	
AR Lag 1	-0.99999	13.44787	-0.99972	17.33195	MA Lag 1
AR Lag 2	0.00000	0.05435	0.00000	0.03460	MA Lag 2
AR Lag 3	0.00000	13.86321	0.00000	13.85744	MA Lag 3
AR Lag 4	0.00000	0.00998	0.00000	0.05520	MA Lag 4
AR Lag 5	0.00000	10.28821	0.00000	9.64858	MA Lag 5
AR Lag 6	0.00000	7.80154	0.00000	8.44422	MA Lag 6
AR Lag 7	0.00000	0.02622	0.00000	0.01202	MA Lag 7
AR Lag 8	0.00000	0.02548	0.00000	0.01269	MA Lag 8
MA Lag 1	-0.99958	14.63837	-0.99455	20.67528	AR Lag 1
MA Lag 2	0.00000	0.05105	0.00000	0.04727	AR Lag 2

Table 7.3: ARMA Models fit to HTM Predictions of (6,0)- and (0,6)-ARMA Models using "lite" parameter swarming

The results of these experiments can be seen in Tables 7.3 and 7.4. For this experiment we used the null hypothesis that the coefficients fitted to the HTM prediction series would be statistically indistinguishable from the coefficients fit to the ARMA models the HTM was asked to predict. All outputs had differences from the mean

Autoregressive Model			Moving-Average Model		
	HTM Output	Z-Score	HTM Output	Z-Score	
AR Lag 1	0.063	0.79751	0.897	15.55602	MA Lag 1
AR Lag 2	0.000	0.05435	0.000	0.03460	MA Lag 2
AR Lag 3	0.000	13.77064	0.000	13.89143	MA Lag 3
AR Lag 4	0.000	0.00998	0.000	0.05520	MA Lag 4
AR Lag 5	0.000	10.22462	0.000	9.73127	MA Lag 5
AR Lag 6	0.000	7.60282	0.000	8.54332	MA Lag 6
AR Lag 7	0.000	0.02622	0.000	0.01202	MA Lag 7
AR Lag 8	0.000	0.02548	0.000	0.01269	MA Lag 8
MA Lag 1	0.000	0.06782	-0.817	17.05749	AR Lag 1
MA Lag 2	0.000	0.05105	0.000	0.04727	AR Lag 2

Table 7.4: ARMA Models fit to HTM Predictions of (6,0)- and (0,6)-ARMA Models using "full" parameter swarming

above 13 sigma which allows us to conclude with very high statistical significance (p approximately zero) that the observed output is not drawn from the sample of outputs of the `arma_order_select_ic` and `fit` function on instances of the true model, rejecting the null hypothesis. This evidence suggests that one-layer HTM networks may not model noisey sequences derived from ARMA models as the same ARMA models internally.

Chapter 8

Further Work

Directly related to this work, there are still questions of whether the prediction outputs of Hierarchical Temporal Memory networks are more similar to the input models for certain classes of ARMA models than others. For example, for certain orders or coefficients an HTM's output may resemble the model much more closely. It would also be interesting to find better ways of standardizing ARMA models to find optimal parameters which avoid the problems introduced by scaling by a constant factor. This work can also be extended to include ARIMA models in the hopes of better understanding how Numenta's Cortical Learning Algorithm interacts with stochastic time series. Additionally, all of the work done in this thesis was on a single-layer network and it would be informative to explore these questions with large configurations to see how that affects results.

It would be interesting to investigate if other encoders or classifiers would have better results for time series analysis as well. Hierarchical Temporal Memory has so far gone through a couple of iterations of encoders and classifiers, and scalar values are a noted problem for HTM networks. In communications with Scott Purdy of Numenta,

he commented on the difficulting in choosing the correct parameters for encoding and decoding, saying "the decoding value is ambiguous within this resolution. In our prediction problems, we find that the right resolution is an important choice in minimizing the prediction error. If the encoding is too course, the predictions can't be reconstructed precisely enough. If they are too fine, the model struggles to learn the temporal patterns and overfits." It could also be possible that sparse distributed codes are not suited for this task, meaning cortical learning algorithms for time series prediction are better off using a different encoding scheme that encodes more information.

Bibliography

[1] C. Adams and R. Franzosa, *Introduction to Topology: Pure and Applied*, Pearson Prentice Hall, Upper Saddle River, NY, 2008.

[2] S. Ahmad and J. Hawkins, *Properties of sparse distributed representations and their application to hierarchical temporal memory*, arXiv preprint arXiv:1503.07469, (2015).

[3] S. Ahmad and J. Hawkins, *How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites*, ArXiv e-prints, (2016).

[4] F. Åslin, *Evaluation of hierarchical temporal memory in algorithmic trading*, Master's thesis, Institutionen för Datavetenskap, 2 2010.

[5] A. Banks, J. Vincent, and C. Anyakoha, *A review of particle swarm optimization. part i: background and development*, Natural Computing, 6 (2007), pp. 467–484.

[6] G. E. P. Box and G. M. Jenkins, *Time Series Analysis: Forecasting and Control*, John Wiley & Sons, Inc., Hoboken, N.J., 5 ed., 2016.

[7] B. G. Buchanan, *A (very) brief history of artificial intelligence*, AI Magazine, 26 (2005), pp. 53–60.

[8] F. Byrne, *Random distributed scalar encoder*. http://fergalbyrne.github.io/rdse.html, 7 2014. Online; accessed 27-April-2018.

[9] J. Connell and K. Livingston, *Four paths to ai*, Frontiers in artificial intelligence and applications, 171 (2008), p. 394.

[10] Y. Cui, S. Ahmad, and J. Hawkins, *Continuous online sequence learning with an unsupervised neural network model*, Neural Computation, 28 (2016), pp. 2474–2504. PMID: 27626963.

[11] ——, *The htm spatial pooler—a neocortical algorithm for online sparse distributed coding*, Frontiers in Computational Neuroscience, 11 (2017), p. 111.

[12] L. N. de Castro, *Fundamentals of natural computing: an overview*, Physics of Life Reviews, 4 (2007), pp. 1 – 36.

[13] A. Dillon, *Sdr classifier*. http://hopding.com/sdr-classifier, 2016. Online; accessed 9-April-2018.

[14] R. Eberhart and J. Kennedy, *A new optimizer using particle swarm theory*, in MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science, IEEE.

[15] P. Gabrielsson, R. König, and U. Johansson, *Evolving hierarchical temporal memory-based trading models*, in Proceedings of the 16th European Conference on Applications of Evolutionary Computation, EvoApplications'13, Berlin, Heidelberg, 2013, Springer-Verlag, pp. 213–222.

[16] M. Galetzka, *Intelligent predictions: an empirical study of the cortical learning algorithm*, Master's thesis, University of Applied Sciences Mannheim, 2014.

[17] D. George and J. Hawkins, *Towards a mathematical theory of cortical micro-circuits*, PLOS Computational Biology, 5 (2009), pp. 1–26.

[18] L. Gugerty, *Newell and simon's logic theorist: Historical background and impact on cognitive modeling*, Proceedings of the Human Factors and Ergonomics Society Annual Meeting, 50 (2006), pp. 880–884.

[19] J. Hawkins and S. Ahmad, *Why neurons have thousands of synapses, a theory of sequence memory in neocortex*, Frontiers in Neural Circuits, 10 (2016), p. 23.

[20] J. Hawkins, S. Ahmad, and Y. Cui, *A theory of how columns in the neocortex enable learning the structure of the world*, Frontiers in Neural Circuits, 11 (2017), p. 81.

[21] J. Hawkins, S. Ahmad, and D. Dubinsky, *Hierarchical temporal memory including htm cortical learning algorithms*, (2011).

[22] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin, *Biological and machine intelligence (bami)*. Initial online release 0.4, 2016.

[23] J. Hawkins and S. Blakeslee, *On Intelligence*, Times Books, New York, NY, USA, 2004.

[24] T. Jones, *Artificial Intelligence: A Systems Approach with CD*, Jones and Bartlett Publishers, Inc., USA, 1st ed., 2008.

[25] A. Lavin and S. Ahmad, *Evaluating real-time anomaly detection algorithms - the numenta anomaly benchmark*, CoRR, abs/1510.03336 (2015).

[26] J. Munkres, *Topology*, Prentice Hall, Upper Saddle River, NJ, 2 ed., 2000.

[27] Numenta, *Advanced nupic programming*, (2008).

[28] ——, *Principles of hierarchical temporal memory (htm): Foundations of machine intelligence*, October 2014.

[29] ——, *Numenta research: Key discoveries in understanding how the brain works*, August 2018.

[30] R. C. O'Reilly and Y. Munakata, *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*, MIT Press, Cambridge, MA, USA, 1st ed., 2000.

[31] S. Purdy, *Encoding data for HTM systems*, CoRR, abs/1602.05925 (2016).

[32] S. Seabold and J. Perktold, *Statsmodels: Econometric and statistical modeling with python*, in 9th Python in Science Conference, 2010.

[33] J. R. Searle, *Mind design*, MIT Press, Cambridge, MA, USA, 1985, ch. Minds, Brains, and Programs, pp. 282–307.

[34] A. M. TURING, *I.—COMPUTING MACHINERY AND INTELLIGENCE*, Mind, LIX (1950), pp. 433–460.

[35] A. Vivmond, *Utilizing the htm algorithms for weather forecasting and anomaly detection*, Master's thesis, University of Bergen, 2016.

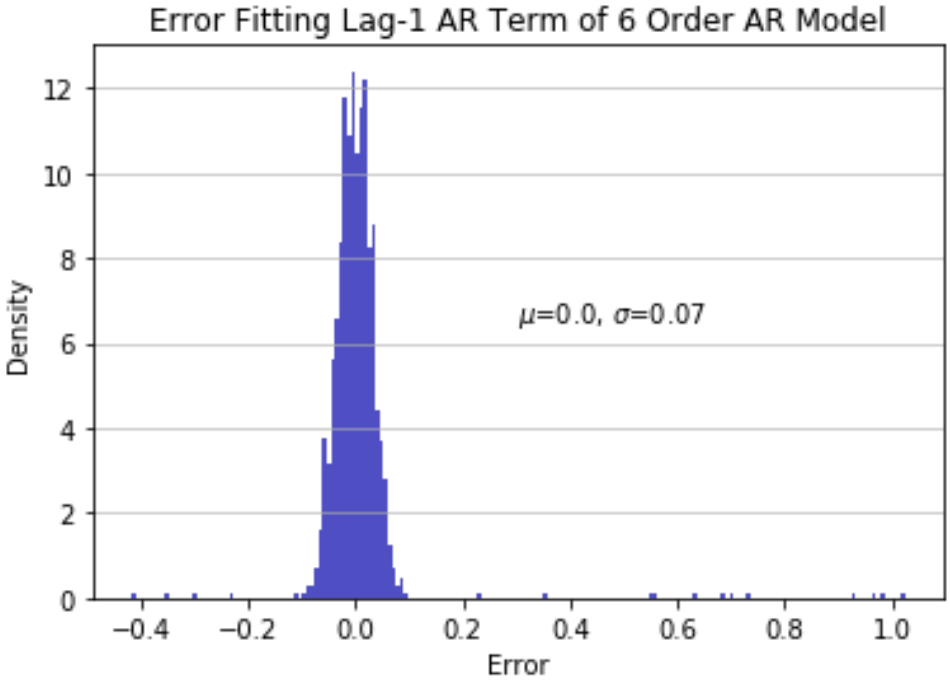
[36] F. D. S. Webber, *Semantic folding theory and its application in semantic fingerprinting*, CoRR, abs/1511.08855 (2015).

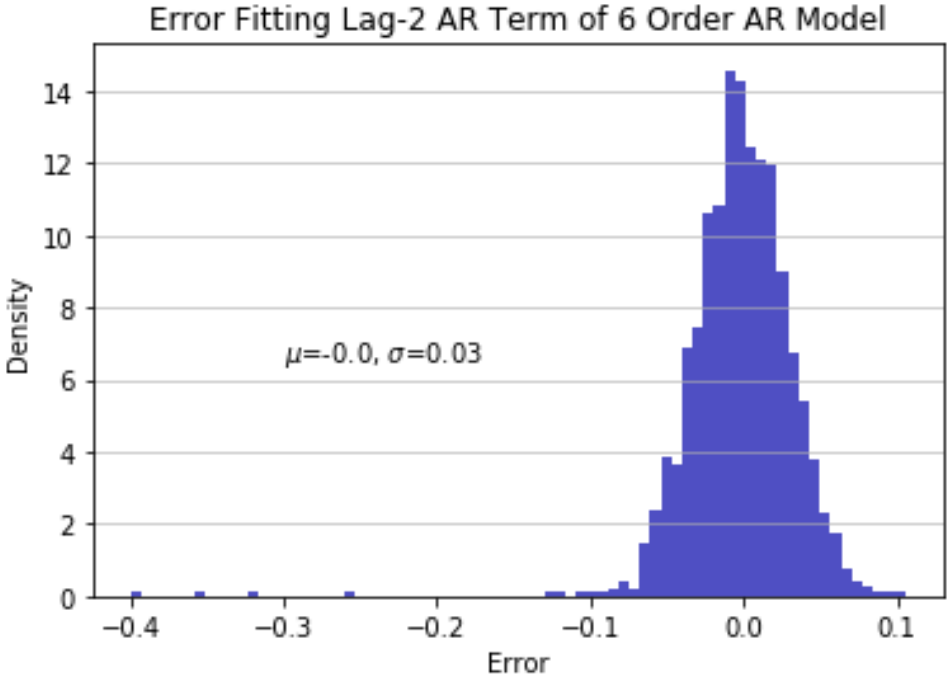
Appendices

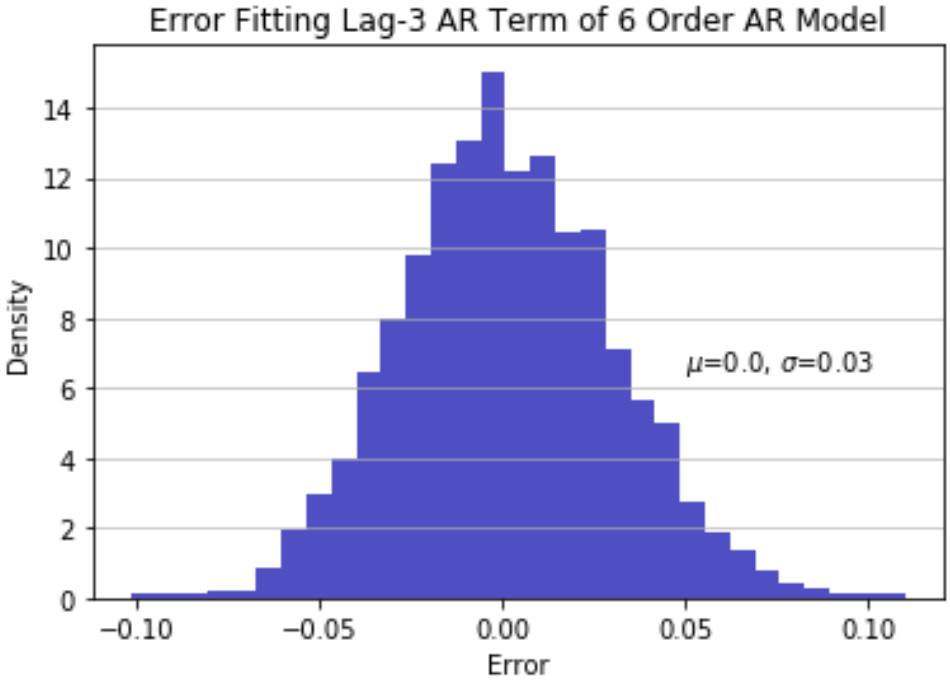
Appendix A

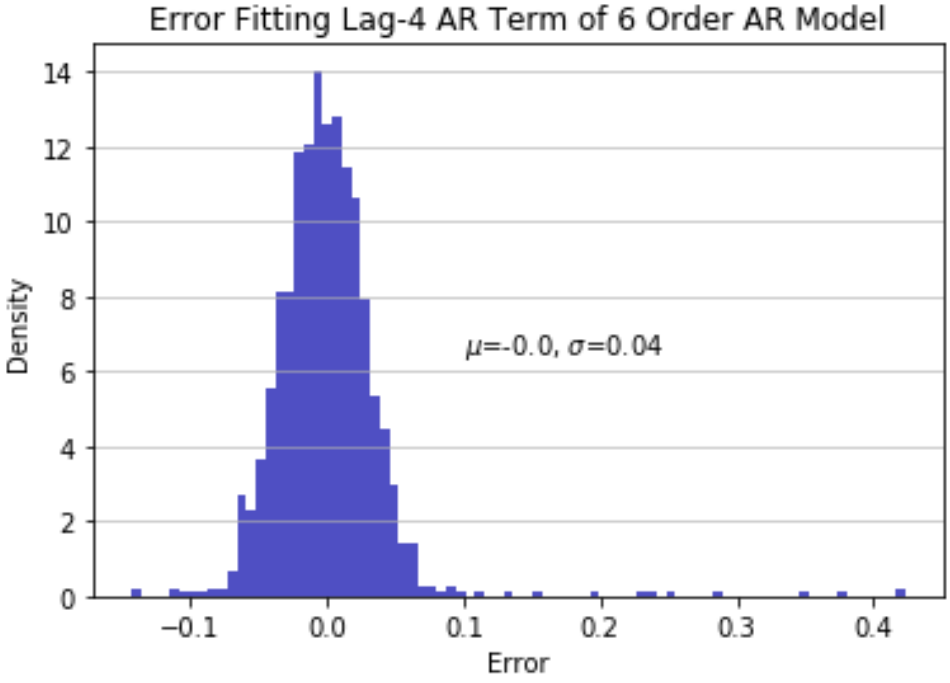
Errors Fitting Coefficients of Models

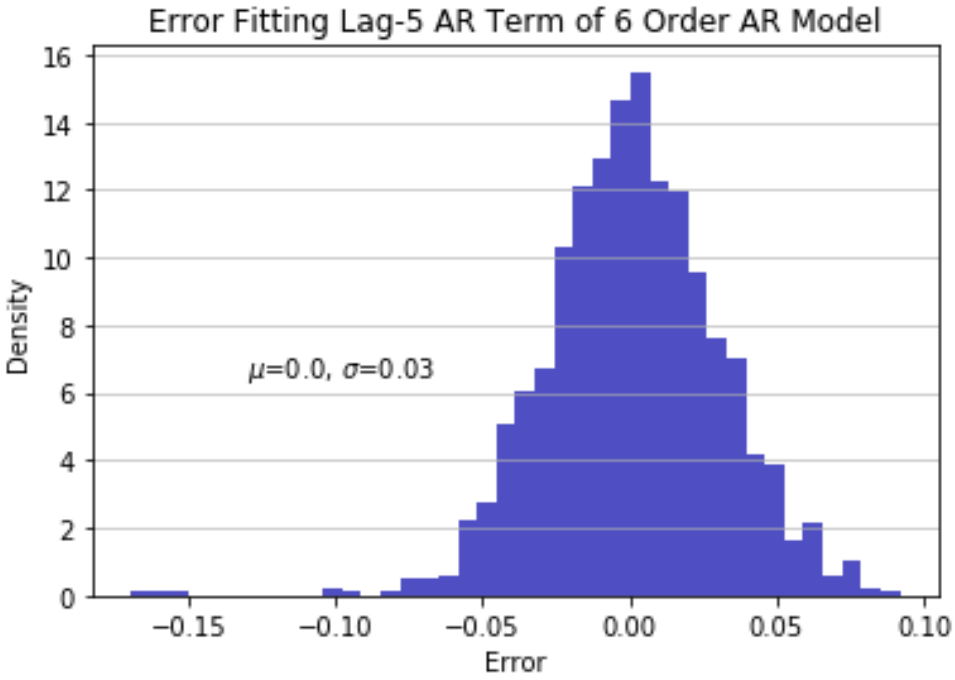
As discussed in Section 6.1.3, below are histograms for the difference between model coefficients and coefficients produced by `fit` for 1,500 instances of AR and MA models with lag polynomials of [0.0, 0.0, 0.4, 0.0, 0.3, 0.3].

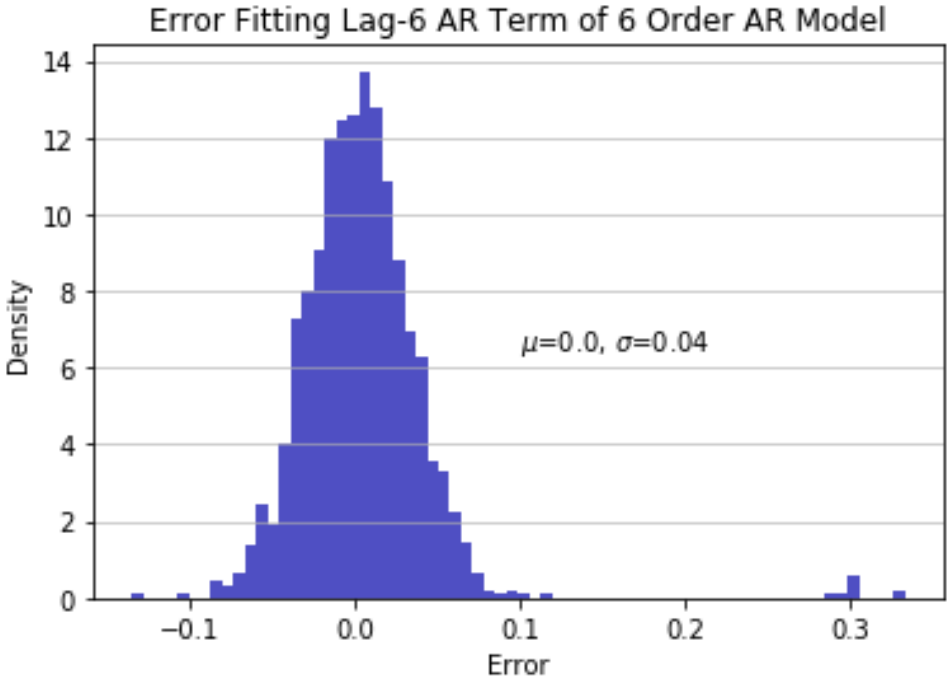

Error Fitting Lag-1 AR Term of 6 Order AR Model
Density
Error
μ=0.0, σ=0.07
0
2
4
6
8
10
12
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
1.0

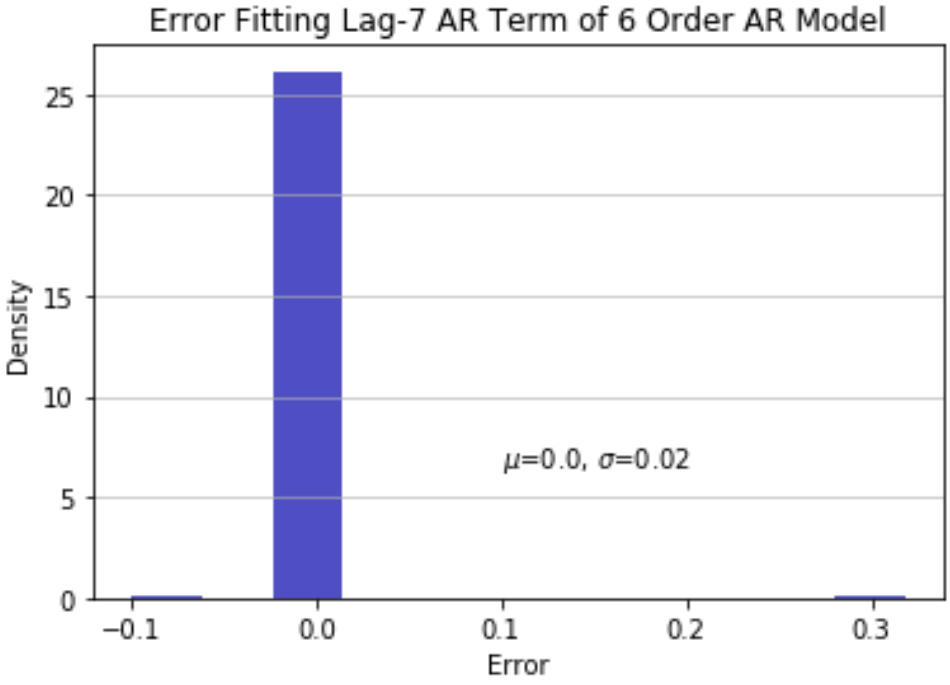

Error Fitting Lag-2 AR Term of 6 Order AR Model
Density
Error
μ=-0.0, σ=0.03
0
2
4
6
8
10
12
14
−0.4
−0.3
−0.2
−0.1
0.0
0.1

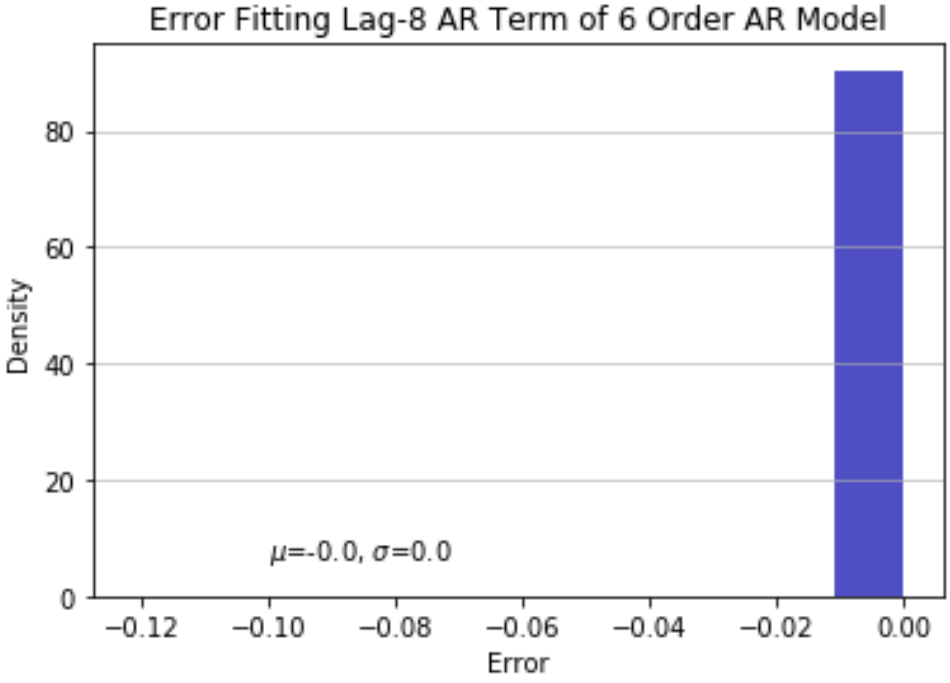

Error Fitting Lag-3 AR Term of 6 Order AR Model
Density
Error
μ=0.0, σ=0.03
0
2
4
6
8
10
12
14
−0.10
−0.05
0.00
0.05
0.10

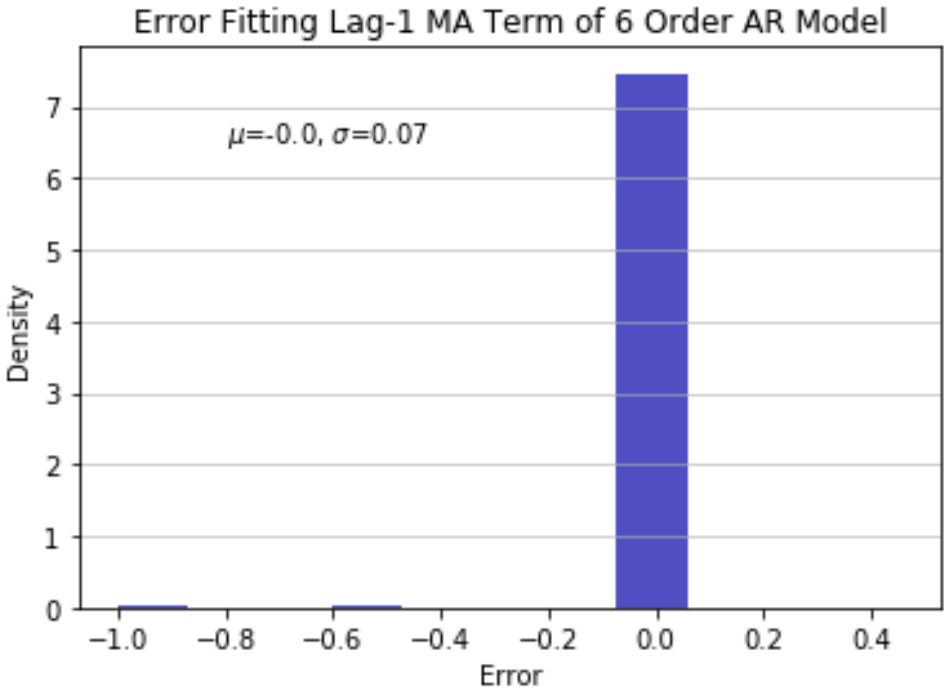

Error Fitting Lag-4 AR Term of 6 Order AR Model
Density
Error
μ=-0.0, σ=0.04
0
2
4
6
8
10
12
14
−0.1
0.0
0.1
0.2
0.3
0.4

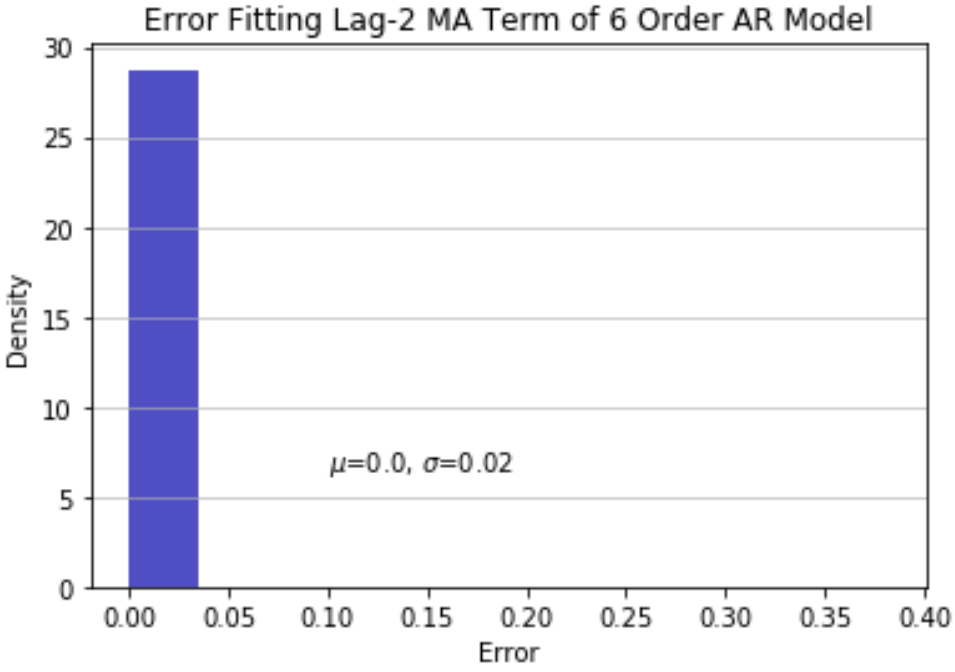

Error Fitting Lag-5 AR Term of 6 Order AR Model
Density
Error
μ=0.0, σ=0.03
0
2
4
6
8
10
12
14
16
−0.15
−0.10
−0.05
0.00
0.05
0.10

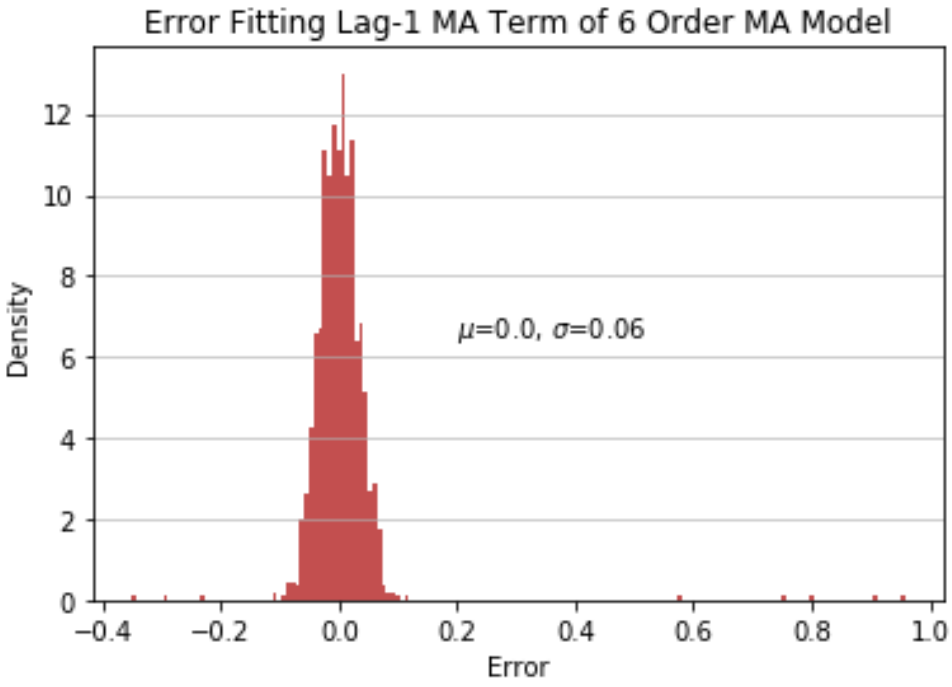

Error Fitting Lag-6 AR Term of 6 Order AR Model
Density
Error
μ=0.0, σ=0.04
0
2
4
6
8
10
12
14
−0.1
0.0
0.1
0.2
0.3


Error Fitting Lag-7 AR Term of 6 Order AR Model
Density
25
20
15
10
5
0
μ=0.0, σ=0.02
−0.1
0.0
0.1
0.2
0.3
Error

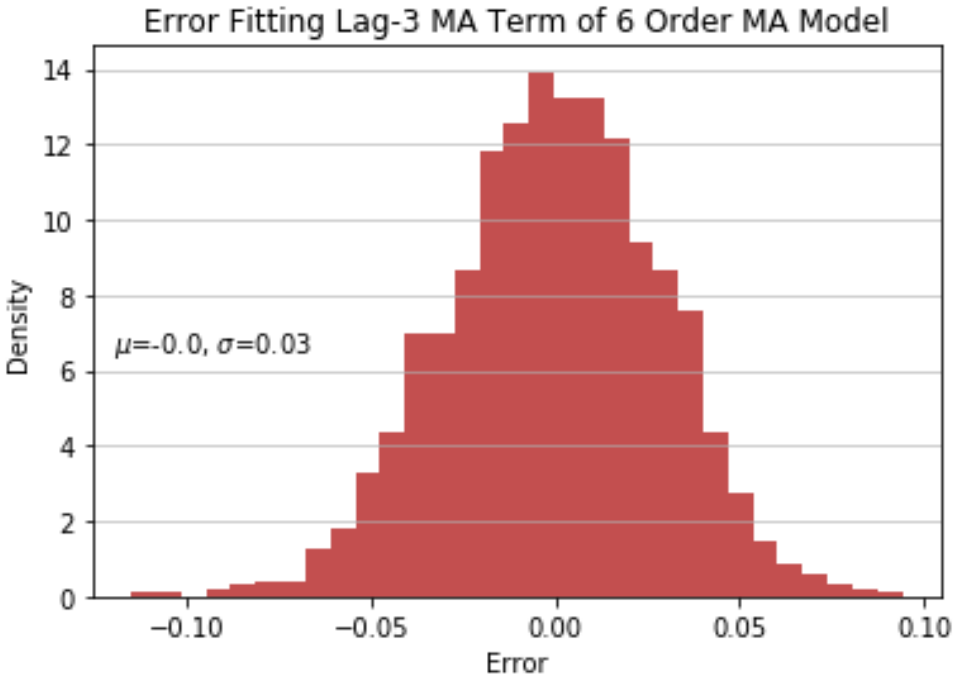

Error Fitting Lag-8 AR Term of 6 Order AR Model
Density
80
60
40
20
0
μ=-0.0, σ=0.0
−0.12
−0.10
−0.08
−0.06
−0.04
−0.02
0.00
Error

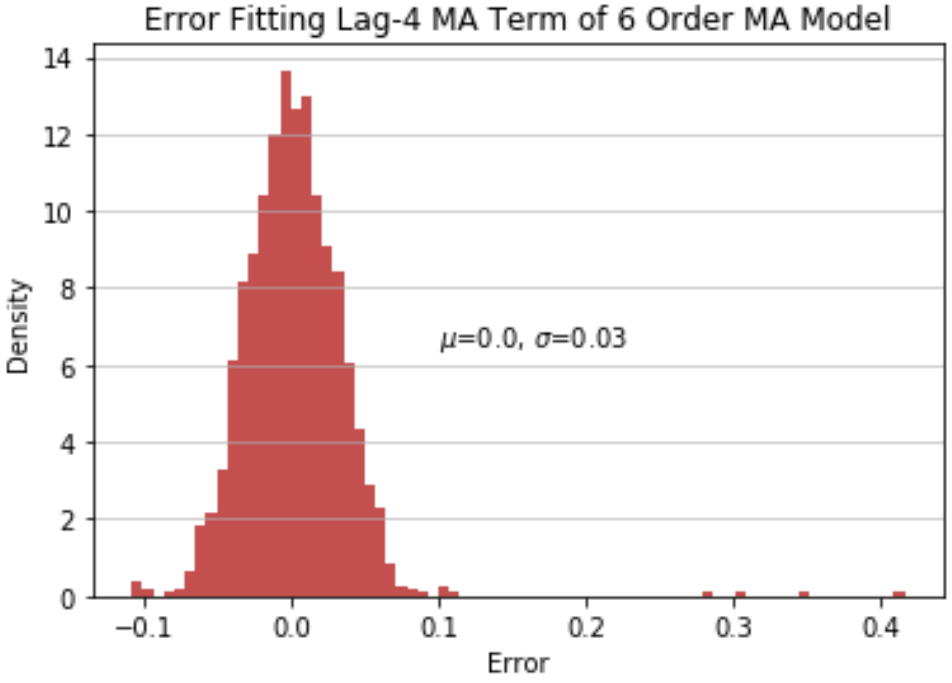

Error Fitting Lag-1 MA Term of 6 Order AR Model
μ=-0.0, σ=0.07
Density
Error
0
1
2
3
4
5
6
7
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4

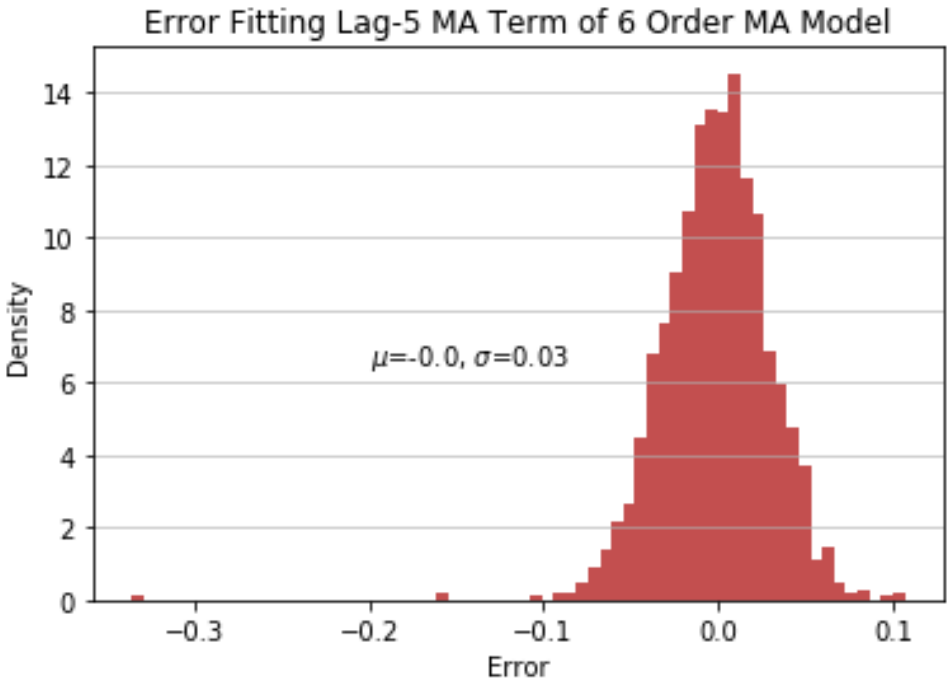

Error Fitting Lag-2 MA Term of 6 Order AR Model
μ=0.0, σ=0.02
Density
Error
0
5
10
15
20
25
30
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40


Error Fitting Lag-1 MA Term of 6 Order MA Model
Density
Error
μ=0.0, σ=0.06
0
2
4
6
8
10
12
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
1.0


Error Fitting Lag-2 MA Term of 6 Order MA Model
Density
Error
μ=-0.0, σ=0.04
0
2
4
6
8
10
12
14
−0.8
−0.6
−0.4
−0.2
0.0

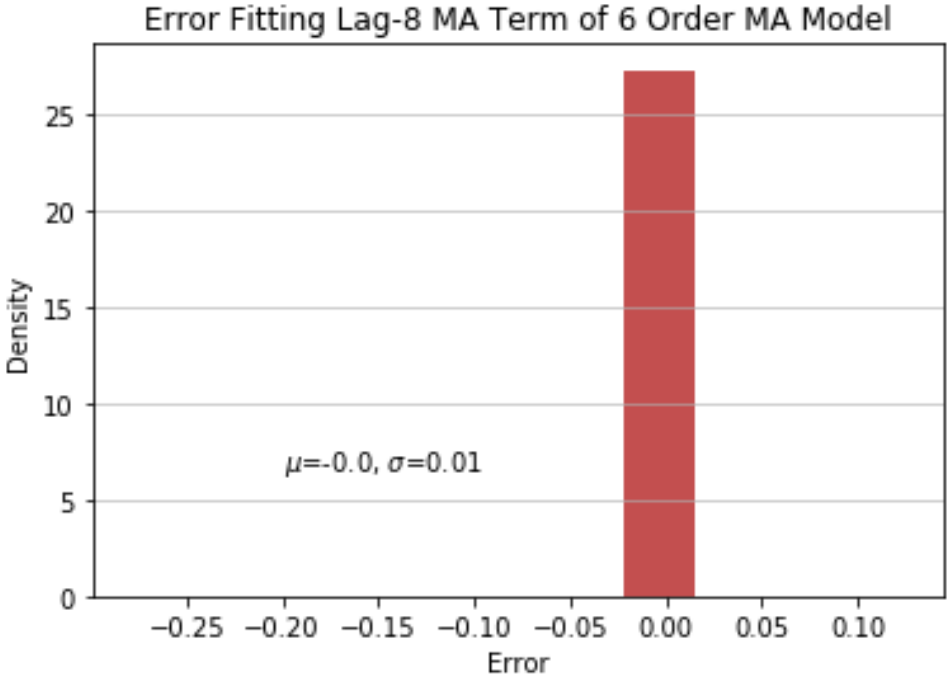

Error Fitting Lag-3 MA Term of 6 Order MA Model
Density
Error
μ=-0.0, σ=0.03
0
2
4
6
8
10
12
14
−0.10
−0.05
0.00
0.05
0.10


Error Fitting Lag-4 MA Term of 6 Order MA Model
Density
Error
μ=0.0, σ=0.03
0
2
4
6
8
10
12
14
−0.1
0.0
0.1
0.2
0.3
0.4


Error Fitting Lag-5 MA Term of 6 Order MA Model
Density
Error
μ=-0.0, σ=0.03
0
2
4
6
8
10
12
14
−0.3
−0.2
−0.1
0.0
0.1


Error Fitting Lag-6 MA Term of 6 Order MA Model
Density
Error
μ=-0.0, σ=0.04
0
2
4
6
8
10
12
−0.1
0.0
0.1
0.2
0.3


Error Fitting Lag-7 MA Term of 6 Order MA Model
Density
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
μ=-0.0, σ=0.01
−0.2
−0.1
0.0
0.1
0.2
0.3
Error


Error Fitting Lag-8 MA Term of 6 Order MA Model
Density
25
20
15
10
5
0
μ=-0.0, σ=0.01
−0.25
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
Error


Error Fitting Lag-1 AR Term of 6 Order MA Model
μ=-0.0, σ=0.05
Density
Error
0
2
4
6
8
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2


Error Fitting Lag-2 AR Term of 6 Order MA Model
μ=0.0, σ=0.03
Density
Error
0
2
4
6
8
10
12
0.0
0.2
0.4
0.6
0.8