DEVELOPMENT OF MACHINE LEARNING TECHNIQUES FOR APPLICATIONS IN THE STEEL INDUSTRY

by

Alex Raynor

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Electrical and Computer Engineering

Department of Electrical and Computer Engineering

Hammond, Indiana

May 2020

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Colin P. Elkin, Co-Chair

School of Engineering

Dr. Vijay Devabhaktuni, Co-Chair

School of Engineering

Dr. Li-Zhe Tan

School of Engineering

Approved by:

Dr. Xiaoli Yang

This thesis is dedicated to my dude Bert and to my parents for always supporting me.

ACKNOWLEDGMENTS

Many people are directly responsible for the research detailed in this thesis, and I would just like to show some appreciation to them for all that they have done.

I would firstly like to thank both Dr. Vijay Devabhaktuni and Dr. Colin P. Elkin for allowing me to have the opportunity to work on this project and become a graduate research assistant. It has been such a great learning experience for me, and the knowledge I have gained from this research is sure to benefit me in my future career pursuits.

I would also like to extend my thanks to ArcelorMittal for sponsoring this project. Without them working with Purdue University Northwest, this project would have never even been available for me to work on and gain experience from.

Many thanks are in order to Rajat Bathla and the rest of the team over at ArcelorMittal. Their knowledge and expertise on the steel manufacturing process was invaluable to me when I was seeking answers on application-specific questions. Moreover, I would like to personally thank Rajat for organizing to meet with me for updates and research suggestions on a regular basis. His direct input helped vastly in deciding avenues to pursue for research on this project.

I would like to thank Dr. Li-Zhe Tan for his input on topics relating to this research. He was willing to give advice and guidance even though he was not directly involved with the project. I continue to extend my gratitude even further to Dr. Tan, as he is easily one of the best teachers I have ever had in my entire academic career. He has helped me so much in my time at Purdue Northwest from small things like help with homework, all the way up to providing research and career opportunities. Without his influence, I certainly would not be where I am today.

Finally, I would like to give thanks to Dr. Xiaoli (Lucy) Yang. Without her informing me of the 4+1 graduate degree option at Purdue Northwest, I might not have ended up going to graduate school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

For a long time, the collection of data through sensors and other means was seen as inconsequential. However, with the somewhat recent developments in the areas of machine learning, data science, and statistical analysis, as well as in the rapid growth of computational power being allotted by the ever-expanding computer industry, data is not just being seen as secondhand information anymore. Data collection is showing that it currently is and will continue to be a major driving force in many applications, as the predictive power it can provide is invaluable. One such area that could benefit dramatically from the use of predictive techniques is the steel industry. This thesis applied several machine learning techniques to predict steel deformation issues collectively known as the hook index problem [1].

The first machine learning technique utilized in this endeavor was neural networking. The neural networks built and tested in this research saw the use of classification and regression prediction models. They also implemented the algorithms of gradient descent and adaptive moment estimation. Through the employment of these networks and learning strategies, as well as through the line process data, regression-based networks made predictions with average percent error ranging from 106-114%. In similar performance to the regression-based networks, classification-based networks made predictions with average accuracy percentage ranges of 38-40%.

To remedy the problems relating to neural networks, Bayesian networking techniques were implemented. The main method that was used as a model for these networks was the Naïve Bayesian framework. Also, variable optimization techniques were utilized to create well-performing network structures. In the same vein as the neural networks, Bayesian networks used line process data to make predictions. The classification-based networks made predictions with average accuracy ranges of 64-65%. Because of the increased accuracy results and their ability to draw causal reasoning from data, Bayesian networking was the preferred machine learning technique for this research application.

1. INTRODUCTION

One of the first steps often taken in understanding an unknown or undocumented process is the measurement of certain aspects and features relating to the not yet understood process itself. However, measurements taken in this fashion are sometimes only seen as supplementary information to be called upon in a reactionary manner when the process starts to defy expectations, or when it is not apparent for what the measurements could be used in order to improve the given process. This thesis strives to reevaluate these thought processes by suggesting that data should be used for proactive implementations like prediction algorithms.

In this day and age, it is very easy and affordable to use measurement and sensor technology to gain data and insight. Moreover, with the current advancements in computational technologies, as well as in the advancements of machine learning, statistical analysis, and data science theories, there is no better time to start using predictive algorithms than right now. These ideas can be used to not only enhance things like manufacturing processes but also to foresee and correct potential problems that have yet to occur. This was the basis behind the idea of using different machine learning techniques in the steel rolling process at ArcelorMittal.

The steel rolling and manufacturing process at ArcelorMittal has seen the implementation of many different sensors that collect information and data on the steel and tools used to roll the steel. Currently, this data is being used for other critical processes, but it has yet to address an issue in the steel rolling process dubbed by representatives from ArcelorMittal as the hook index problem [1]. The hook index problem that rolls of steel can encounter during the line process is described as an irregular elongation of the ends of the steel that, when rolled into coils at the end of the process, jut from the center of the donut-like shape of the coil. These jutting ends and irregular coil shapes make steel storage, transportation, and other process difficult to manage.

The solutions that are currently in practice by ArcelorMittal require reprocessing time and waste money and resources. This waste of valuable resources could ultimately be saved if measures could be put in place that could predict when steel rolls are likely to experience hook indexes. That is

the gap that this research has sent out to fill through the use of different machine learning techniques and other data science theory implementations.

Detailed in this thesis are the steps taken to approach and subsequently provide information for potential solution implementations for the hook index problem [1]. The two major machine learning techniques evaluated during this research were neural networks and Naïve Bayesian networks. These techniques use data provided by ArcelorMittal and set a value representing the hook index as the output to be predicted by these network structures. The findings, results, and suggestions collected from this research are detailed in the later chapters. The actual layout and chapter structure of the thesis after this introductory chapter is explained thusly. Chapter 2 contains a comprehensive review of the ideas and previous works studied in preparation of this thesis. Chapter 3 covers the methods that went into implementing and testing the different network structures. Chapter 4 puts forward the results received from the tested networks. Chapter 5 discusses the aforementioned results, presents future work ideas and potential solutions, and draws final conclusions.

2. LITERATURE REVIEW

In an effort to gain background knowledge on the topics covered in this thesis, as well as view what kinds of machine learning techniques have been implemented in similar applications to this research, different literature pieces and topics were referenced and reviewed. More specifically, topics referring to machine learning in steel applications and explanations of machine learning techniques were researched.

Several literary works were reviewed for machine learning applications in the steel industry. The topics that were covered by these papers were the optimizations of amount of steel produced by the manufacturing process of a mill [2] and the quality assurance and control of steel being manufactured [3]. The machine learning processes that were used in these reviewed works are the basis of discussions in the following paragraphs.

Starting with the first mentioned topic, researchers trying to optimize and increase steel production did so using a multitude of different machine learning techniques. To be more specific about the techniques covered in that particular research, the techniques that were covered were random forest methods, artificial neural networks, support vector machines, and dynamic evolving neuro-fuzzy inference systems [2]. All of these techniques worked in such a way to which they took in inputs from the process that they were measuring and predicted a single output through the use of a regression-based learning method. Moreover, these techniques all used mean squared error, absolute percent error, and mean absolute percent error cost functions to train the given machine learning techniques. The particular methods behind each machine learning technique will be discussed for each technique individually.

For the random forest method, the process is described as an implementation of multiple decision tree methods working with different input data subsections [2, 4]. The groups of variables used in the subsection of each tree is determined randomly and gets trained to improve by a training set of data. Each tree grows by using a version of the method known as classification and regression tree method. The basis for this method comes in the form of making binary decisions at each node in the tree structure until a leaf node is reached [4, 5]. However, in the random forest method, each

tree implements elements of randomness into the tree building and data feature selection process. The collection of these random trees is what makes this method a random forest implementation. In the reviewed literature, the researchers used a random forest of one hundred trees and split input variables in groups of three [2].

The next method to be discussed is the artificial neural network technique. In the literature, the method utilizes multiple layers of a network that move input data through the network in a forward propagation process [2, 6]. The gradient descent method of learning is the most widely used, but other types were implemented in the research as well. These other algorithms were the quasi-Newton, scaled conjugate gradient, and Levenberg–Marquardt (LM) algorithms [6-9]. The quasi-Newton method converges to network minimum points more quickly than the other methods but does not take network accuracy into account as well as the other methods. The scaled conjugate gradient method is described in an inverse fashion to the quasi-Newton method, in that it considers network accuracy more carefully but reaches local or global minimum points more slowly. To balance these two training ideals, the LM method was introduced into the research to attain better network accuracies and converge to error minimums more quickly [2].

Dynamic evolving neuro-fuzzy inference systems are a combination of neural networking techniques and fuzzy systems [2, 10]. These have five layers. The first layer is the data input layer, the second is the fuzzy input layer, the third is the rule-based layer, the forth is the fuzzy output layer, and the fifth and final layer is the output prediction layer. The method is trainable in both online and offline circumstances. The online version implements a maximum distance-based clustering method to change the parameters of the membership function of the network, while the offline version uses a constrained version of this algorithm. The rule layer implements a Takagi–Sugeno–Kang fuzzy inference engine, and the output in the output layer is the weighted average of the outputs of the rule layer [2, 11].

The final machine learning method discussed in the cited research is the support vector machines technique. It is a comprehensive algorithm for optimizing models based on both regression and classification learning techniques [2, 12]. The classification support vector machines are detailed to work in two steps. The first step has the input data to the method reassigned to a larger-ordered

data space, while the second has the learning algorithm segment hyperplanes in the larger-ordered data space. Regression support vector machines work in largely the same way, but they try to achieve an optimal regression of the data on a space that has a maximum flat property and a minimum error property [13].

The team conducting the research in this literary work concluded that support vector regression was the best method for optimizing steel output processes [2]. They detailed that it was computationally faster than the dynamic evolving neuro-fuzzy inference systems method, and it provided relationship explanations between data inputs and prediction outputs, which were something that artificial neural networks and random forest methods could not do.

Moving to the second steel application topic, the objective over the research in this literature work was to enhance quality control measures through the implementations of machine learning techniques [3]. This was achieved by using several different learning techniques. The techniques detailed in this research are linear regression, ridge regression, lasso regression, elastic net, support vector machines, kernel ridge regression, K-nearest neighbors, random forest, gradient boosting decision tree, light gradient boosting machine, and extreme gradient boosting.

This literary work does not detail the inner workings of the machine learning processes used, but it does list a process called ensemble learning in which multiple techniques are combined in order to create a more powerful prediction method [3, 14]. This method is said to work in two different cases. The first case, known as the averaging ensemble model, links the support vector machine, kernel ridge regression, gradient boosting decision tree, and extreme gradient boosting techniques together through an averaging process. The second case, known as the stacking ensemble method, links the support vector machine, kernel ridge regression, light gradient boosting machine, and extreme gradient boosting techniques together through a technique stacking process [3, 14-17].

The results received from each method were gauged on a few criteria. One of the criteria was the use of the R-Square expression to quantify the prediction capabilities of a method [3, 18]. Another was to use the root mean square error expression to find a standard deviation of the error received

from each tested method [3, 19]. The ultimate findings of the research were that the averaging ensemble method performed best in the application that is was used for.

When comparing the research this thesis has done to previous works, some similarities can be seen in the methods applied to solve the problem at hand. One such similarity is the fact that some previous works were shown to use artificial neural networking techniques in their applications. Delving deeper into this similarity, previous works used neural networks that were trained from regression-based learning techniques in order to reach a single output prediction. Other, more general, similarities can be drawn in the fact that many previous works used both regression and classification techniques to predict the application-specific issue.

Not all of the research done in this thesis has seen implementations elsewhere. One of the first diverging factors between this thesis and previous works is that the problem this research aims to solve is a specific steel deformity issue [1]. Most previous works reviewed for the purposes of this research aimed to apply machine learning techniques to larger-scale problems and processes [2, 3]. Another deviation that this research achieved is the use and development of Bayesian learning techniques to enhance steel making processes. To explain further, this thesis put into practice the ideas of brute force and survival variable testing to select input variables that are well suited for the specific problem at hand. All of the methods applied in this thesis are detailed in Chapter 3.

3. METHODS

This chapter will cover in detail the theory and overall methodology used to test, train, and complete the different machine learning processes that were used throughout the duration of the research. The two main machine learning areas that were implemented extensively were neural networking and Bayesian networking. Neural networking, and the theory and applications that surround it, are explained in Section 3.1. Bayesian networking, and the theory and applications that surround it, are presented in Section 3.2.

3.1 Neural Networking

Neural networking can be succinctly summarized as a mathematical regression theory that revolves around the interconnections between the nodes or neurons of a network [20]. The distinction of calling the nodes of an interconnected network neurons has roots in the study of human brain functions. The physical neurons in the human brain form firing patterns with other neurons in the spaces around them [21]. These firing patterns act as ways to communicate information like thoughts, actions, and behaviors. As these neural patterns or pathways are used more often, the connections between neurons becomes stronger, which directly correlates to learned behaviors in humans. In that regard, it should be no secret that the mathematical process of neural networking functions similarly to the physical process of learning in humans [22]. Having this view in mind is helpful in understanding the topics that will be covered in the coming sections. The coverage will include the topics of forward propagation, backward propagation, network configurations, types of trained networks, data models, and the limitations of neural networking.

3.1.1 Forward Propagation

To provide some background information and a large scale view of the theory for the sake of clarity, forward propagation is the process by which a neural network processes data that has been input through the input layer of a network [23]. This input information is then manipulated as it travels through the hidden layers, the neurons, and the pathways between the neurons. A graphical depiction of the process being discussed can be seen in Figure 3.1 [24] to help better illustrate the terminology.

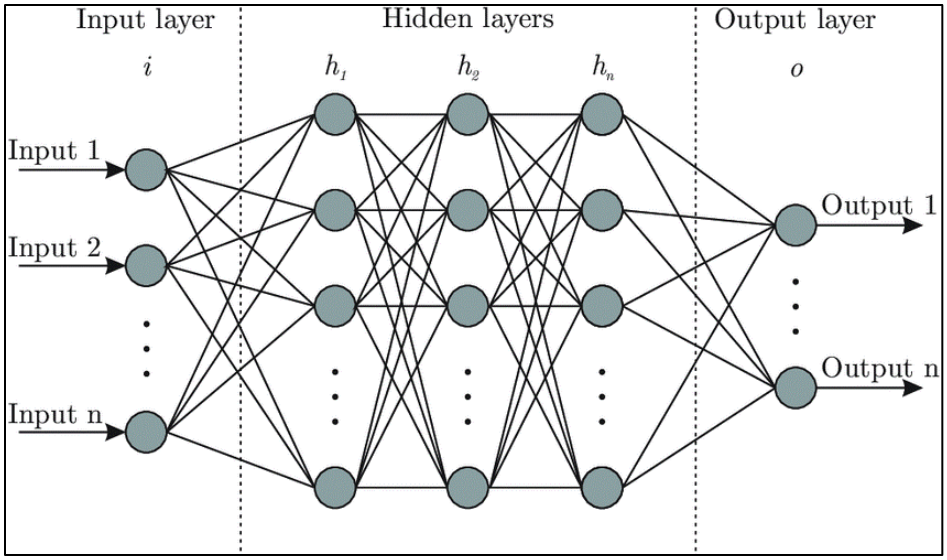

Figure 3.1 Overarching structure of neural networks.

Upon passing through the output layer, the previously input data has now been transformed, through the neural network works equations, into a prediction of a specific network output.

Weight Initialization Techniques

To understand how a neural network calculates and predicts anything, the forward propagation process has to be divided into more elementary operations. To begin, a network will initialize the values it uses for neuron pathway weights and neuron biases [25, 26]. These weight initializations are achieved through the use of a random number generation process that pulls numbers from the standard normal distribution curve for each weight. This process can be optimized beyond the random sampling of the standard normal distribution curve, depending on the type of activation function that is used for neuron activations.

Activation functions are a topic that is discussed later, but the two used in this thesis were the logistic sigmoid and rectified linear unit (ReLU) functions. The formulas associated with optimizing the weight values for the sigmoid and ReLU functions are known as the Xavier and He initialization techniques [23, 27]. The formulas for the Xavier and He initializations are

$$\mathbf{W}_j = \mathbf{R}_j \sqrt{\frac{1}{n_{j-1} n_j}} \qquad \text{(Equation 3.1)}$$

and

$$\mathbf{W}_j = \mathbf{R}_j \sqrt{\frac{2}{n_{j-1} n_j}} \qquad \text{(Equation 3.2)}$$

respectively, where $j = 1,2, \dots, m$, and is the index of the given set of weights that are entering the current network layer, m represents the total number of sets of weights that the network contains in its structure, $\mathbf{W}_j$ is a matrix of all the weight values entering the current network layer that were scaled from the random normal distribution weight value matrix, represented by $\mathbf{R}_j$, and n is the total number of neurons in the indexed network layer. The multiplication of the number of neurons in the previous and current layers of the network aids in providing the equation calculations with the number of weight connections going into the current layer. With the implementation of these initialization techniques for either of the respective activations functions, the weight values of a network should ideally have a mean value of zero and a standard deviation value of one, instead of exploding into large numbers or vanishing into small numbers.

Linear Neuron Outputs

After the weight and bias initialization process, a network will pull in data from its inputs to be used in later calculations. The next of the operations that will be discussed is the equation by which individual neurons calculate linear or intermediary outputs. The intermediary output equation of a network neuron is

$$z_i = \mathbf{w}_i^{\mathrm{T}} \mathbf{x} + b, \qquad \text{(Equation 3.3)}$$

where $i = 1,2,3, \dots, n$, and is an indicator of which neuron in the current layer is being manipulated, n is representative of the total number of neurons in the current layer, z_i represents an individual intermediary or linear output of one specific neuron in the current layer, b represents the bias value of the given neuron whose output is being calculated, and $\mathbf{w}_i$ and $\mathbf{x}$ represent column vectors of size one by the number of neurons in the previous layer [4]. Each vector is filled with

the individual weights or outputs of the neurons in the previous layer to the neuron being manipulated in the current layer. Equation 3.3 can be reduced if an addition weight value and an addition leading one are added to $\mathbf{w}_i$ and **x**, respectively. The resulting expression is [28]

$$z_i = \mathbf{w}_i^{\mathrm{T}}\mathbf{x}. \qquad \text{(Equation 3.4)}$$

Activation Functions

Once the linear output calculation process is completed, the output is fed into another equation called an activation function. This final neuron output equation is

$$y_i = f(z_i), \qquad \text{(Equation 3.5)}$$

where $f(z_i)$ represents the linear output being input into an activation function to calculate the final output of the forward propagation pass of any non-input network neuron, which in this case is represented by y_i. An activation function serves as a method of adding non-linearity into the structure of a network. This is ultimately done to allow for a neural network to be able to create a pseudo, higher-order polynomial or prediction function. This in turn allows a network to learn and predict data much better than any strictly linear function or regression ever could. The whole output calculation process of a neuron can be seen graphically in Figure 3.2 [29].

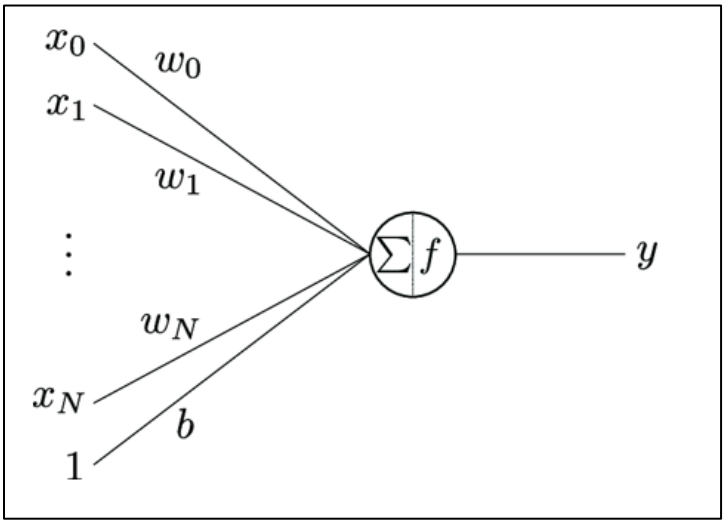

Figure 3.2 Output calculation of a network neuron.

Activation functions are non-linear functions that give neural networks more complexity in order produce better prediction results. However, different activation functions perform different non-linear equations, yielding sometimes vastly different neuron outputs and network firing patterns. Two of the most common activation functions used in network configurations are the logistic sigmoid function and the rectified linear unit (ReLU) function. These functions are

$$y_i = \frac{1}{1 + e^{-z_i}} \qquad \text{(Equation 3.6)}$$

and

$$y_i = max(0, z_i), \qquad \text{(Equation 3.7)}$$

where Equation 3.6 represents sigmoid activation and Equation 3.7 represents ReLU activation [30, 31]. The two activation functions take in the linearly defined output from the current neuron and transform it into a non-linear output. The graphs in Figure 3.3 and Figure 3.4 illustrate how the output ranges of the logistic sigmoid and the rectified linear unit functions are defined, respectively.

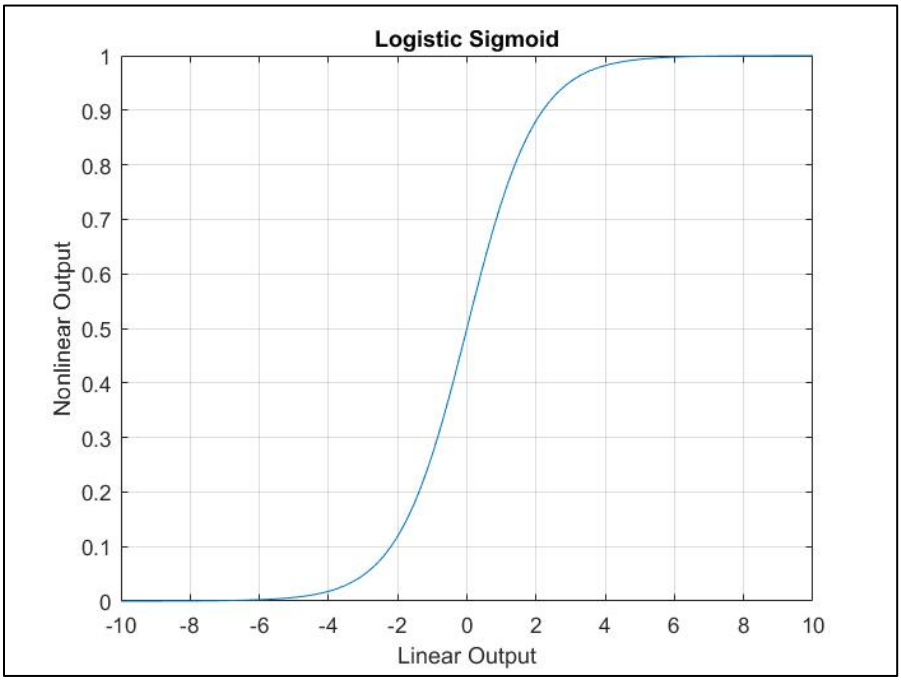

Figure 3.3 Graph of the logistic sigmoid function.

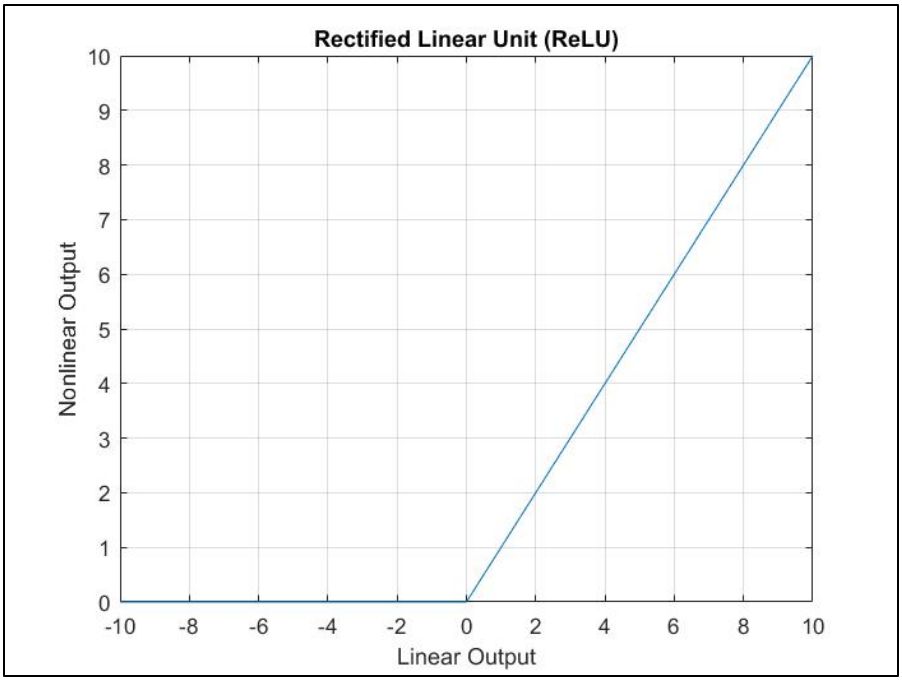

Figure 3.4 Graph of the rectified linear unit (ReLU) function.

Out of the two functions, the logistic sigmoid has been used in many older machine learning and neural network examples because it tended to yield faster network learning rates than some of the other proposed activation functions of that earlier time period [20]. The concept of the logistic sigmoid has also been used in creating the softmax activation function, which can be used as a classifier function in a classification neural network model. The ReLU function was later introduced into the neural network setting as a way to combat some of the shortcomings of the sigmoid function [31]. One such shortcoming is the fact that the sigmoid activation function can produce what is known as a vanishing gradient. The gradient learning method is covered in a later section of this thesis, but this problem can essentially halt the learning process of a neural network due to the fact that the derivative of the sigmoid function will often times calculate values that are close to zero [32]. The ReLU function sidesteps this issue by have a much simpler gradient derivation that is more well defined on the positive x-axis, as well as by creating more sparse network paths since negative outputs can never be activated in a neuron due to the nature of the function definition. This leads to overall better defined neuron patterns without the introduction of more superfluous pathways between neurons.

The Total Network Cost Function

When information travels through a neural network and becomes an output network prediction, the validity of the prediction is called into question. This questioning, or validation process, is done through the use of an error, or cost, function. Error functions are implemented in neural networks, as well as in other machine learning applications, because the use thereof allows for a network to starting learning from the mistakes it makes [20]. How this works is that a set of data will be input into a network to create an output prediction. The network prediction will then be compared to the actual recorded output of a given training data segment. The output that the network predicts is input into an error function to compare against the actual training output. This allows for the neural network to quantify how well or poorly it is predicting outputs. This is done in an effort to help a network correct its inaccurate output predictions.

The error or cost function that was used in this research was the mean squared error function. The mathematic expression used to implement the function into network configurations is

$$J(y_i) = \frac{1}{n}\sum_{i=1}^{n}(\bar{y}_i - y_i)^2\,, \qquad \text{(Equation 3.8)}$$

where y_i is the predicted output value of each neuron in the output layer, and $\bar{y}_i$ is the already known output value of each neuron in the output layer from the dataset for comparison [33]. The function takes the squared error value calculated from the predicted and actual outputs of each neuron, performs a summation on all of the squared error values, and divides by the number of neurons in the output layer of the network. This is done to produce a mean value of the squared errors of the entire output layer, and subsequently, the network as a whole. With a way for the network to gauge how well it is performing, it can now develop methods of improvement through the use of backward propagation.

3.1.2 Backward Propagation

In the same way that a neural network can pass data from inputs to calculate some defined system output, a neural network can also pass error information back through the system to augment

pathway weight values in order to try to minimize output error. A network achieves these error minimizations by using a technique known as gradient descent. To explain the process of how a neural network uses gradient descent to minimize output error in a simplistic sense, a network performs relatively small steps along the slope of its cost function with respect to network weight values until the error value of the whole dataset from which the network is pulling data reaches a minimum point [34]. This process is portrayed graphically in Figure 3.5 [35].

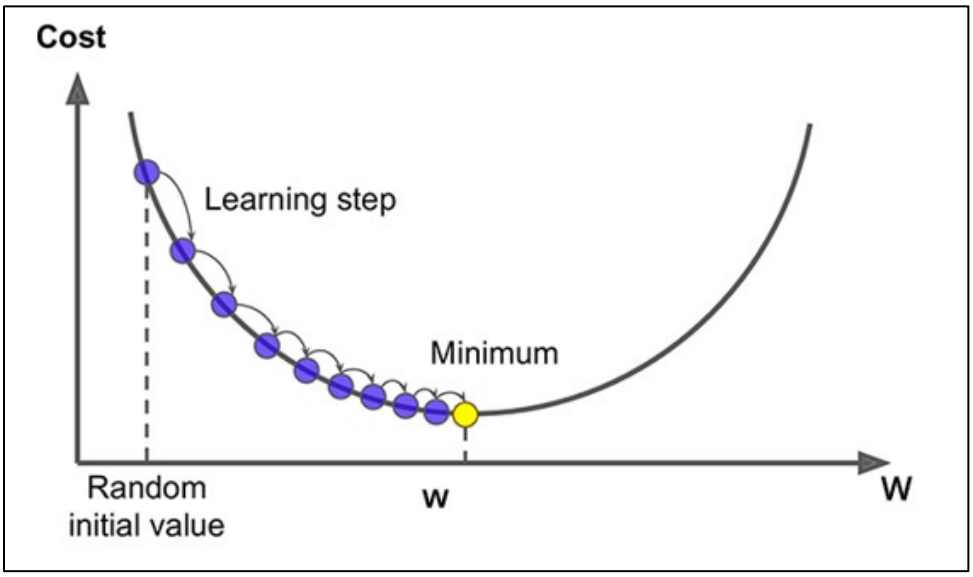

Figure 3.5 The idealized process of the gradient descent algorithm.

The Standard Gradient Descent Algorithm

To dive deeper into the mathematical derivation of the gradient descent algorithm used in network calculations, the concept of a gradient requires elaboration. A gradient is a generalized way of differentiating or finding the slope of a function with respect to multiple dimensions or axes [36]. The idea behind using gradients in the scope of neural network calculations is to relate the error to all of the weights of the network [28]. This is done to directly control and reduce the error that a network produces by manipulating the individual weights of the network.

To start the backward propagation process, the output layer cost to weight gradient function must be derived. This is achieved by first taking the derivative of the network error or cost function with respect to the network output. This derivation changes depending on the use of the specific cost

function that was chosen for the network. The mean squared error cost function was chosen for network implementations in this research, thereby solving as the basis for the further detailed derivatives. The derivative of the mean squared error function with respect to the output of the network is [33]

$$\frac{\partial J}{\partial y_i} = \frac{2}{n}\sum_{i=1}^{n}(\bar{y}_i - y_i)\,. \qquad \text{(Equation 3.9)}$$

As described before, y_i and $\bar{y}_i$ are the predicted and actual output related to each neuron in the output layer of the neural network respectively, and n is the total number of output neurons in the current layer of the network. The numerical calculation of this derivative results in the mean doubled error of the system.

The next step in the process of relating the gradient of the total cost function of the system to the gradient of the weights coming into the output layer is to calculate the derivative of the non-linear output of a neuron with respect to the linear output of the same neuron. In a similar fashion to the error function, this derivative also changes based on the activation function that was used to calculate the non-linear output. The expressions relating to the sigmoid and ReLU activation functions are

$$\frac{\partial y_i}{\partial z_i} = \left(\frac{1}{1+e^{-z_i}}\right)\left(1-\frac{1}{1+e^{-z_i}}\right) \qquad \text{(Equation 3.10)}$$

and

$$\frac{\partial y_i}{\partial z_i} = \begin{cases} 1, & z_i > 0 \\ 0, & z_i \leq 0 \end{cases}, \qquad \text{(Equation 3.11)}$$

respectively [30, 31]. As previously detailed, z_i is the linear or intermediary output of a neuron, and y_i is the non-linear output of the same neuron.

Continuing onto the next step of the derivation, the derivative of the linear output of a neuron with respect to the weights going into said neuron must be expressed. This expression is [37]

$$\frac{\partial z_i}{\partial \mathbf{w}_i} = \mathbf{x}. \qquad \text{(Equation 3.12)}$$

Like before, $\mathbf{x}$ represents a column vector of size one by the number of neurons in the previous layer plus one additional element to account for the bias term of the current neuron. The vector is filled with the output values of the neurons from the previous network layer and an additional leading one. Similarly, $\mathbf{w}_i$ represents a column vector of size one by the number of neurons in the previous layer plus one additional element to account for the bias term of the current neuron. The vector is filled with the weight values of the neurons from the previous network layer and the leading element containing the bias value of the current neuron.

The final step in relating the cost function to the weights coming into each neuron in the output layer is to apply the chain rule to all of the previously calculated derivatives to produce the final relationship expression. This expression is [37]

$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial \mathbf{w}_i}. \qquad \text{(Equation 3.13)}$$

This equation only applies to all of the weight pathways entering the output layer of the network. In fact, applying the chain rule for usage in calculations for the neural network can be easily accomplished by multiplying the values found at each equation step in the derivation process.

Further derivations for layers earlier on in the network can be found by repeating the previously mentioned process. However, to access layers beyond the output layer, one final derivation is required. This derivation is the derivative of the linear output of a neuron with respect to the input vector of the output of each neuron in the previous layer. The operation explaining this process is [37]

$$\frac{\partial z_i}{\partial \mathbf{x}} = \mathbf{w}_i. \qquad \text{(Equation 3.14)}$$

After implementing this calculation, the process is to treat $\mathbf{x}$ in the current layer as all of the output values of the neurons in the previous layer. Then, repeat either Equation 3.10 or 3.11, depending on which activation function is being used, and then use Equation 3.14 if going back another layer is required or use Equation 3.13 to find the derivative of the cost function with respect to the weights of the current network layer at the time. The equation is [37]

$$\frac{\partial J}{\partial \mathbf{W}_l} = \frac{\partial J}{\partial \mathbf{y}_m} \prod_{j=m}^{l+1} \left(\frac{\partial \mathbf{y}_j}{\partial \mathbf{z}_j} \frac{\partial \mathbf{z}_j}{\partial \mathbf{y}_{j-1}} \right) \frac{\partial \mathbf{y}_l}{\partial \mathbf{z}_l} \frac{\partial \mathbf{z}_l}{\partial \mathbf{W}_l}. \qquad \text{(Equation 3.15)}$$

To thoroughly explain the mathematical process expressed in Equation 3.15, j is the current set of network weights that are being indexed, m represents the index of the last set of weights in the neural network and the total number of sets of weights in the network as a whole, and l represents the index of the set of weights that is being related to the cost function of the network. To give an example, if l was equal to m, the set of weights being related to the cost function would be the weights that were entering the output layer of the network. If l was equal to one, the set of weights being related to the cost function would be the weights that were entering the first hidden layer of the network. Next, $\mathbf{z}_j$ and $\mathbf{y}_j$ are vectors of all of the linear and non-linear outputs relating to all indexable layers, and $\mathbf{W}_l$ represents a matrix containing all of the weights associated with all of the pathways entering the layer indexed by l.

With Equation 3.15 defined, a further generalization can be made to finally represent the gradient of the total cost function with respect to the weights of the neural network. This generalized expression is [38]

$$\nabla J(\mathbf{W}) = \left(\frac{\partial J}{\partial \mathbf{W}_1}, \frac{\partial J}{\partial \mathbf{W}_2}, \dots, \frac{\partial J}{\partial \mathbf{W}_m} \right). \qquad \text{(Equation 3.16)}$$

Equation 3.16 contains the expression that allows for each relation between the error function of the network and the set of weights entering a given layer in the network to be vectorized into a gradient vector. $\mathbf{W}$ is used to represent a matrix that contains every single weight contained in the network. The gradient described in Equation 3.16 can finally be used to manipulate the weights of

the neural network through the use of a simple subtraction and gain value. This operation results in the equation [38]

$$\mathbf{W} = \mathbf{W} - \alpha \nabla J(\mathbf{W}). \qquad \text{(Equation 3.17)}$$

The operation detailed in Equation 3.17 shows how the error function minimization stepping process works. $\nabla J(\mathbf{W})$ represents the gradient of the error function with respect to the network weights, and α represents a preset gain known as the learning rate. The gradient matrix is scaled by the learning rate to effectively control the step size of all the network weights as a whole. This approach to implementing how weight step sizes are scaled is seen as the standard or classical way of manipulating the weights of a network.

While the classical concept and implementation of the gradient descent method of changing network weights is revolutionary, it does not come without flaws of its own. The most glaring flaw with this method is the issue of a one size fits all learning rate [39]. In brief, this issue stems from the fact that scaling and stepping all weights by the same gain factor usually results in less than optimal network performances. This idea is solidified through an explanation of picking a learning rate for a network in the standard gradient descent method. If the learning rate chosen was too large for the process, suboptimal minimum points might be found for most of the total cost to weight curves that have been established. In a worst case scenario, networks could even train to be more error prone due to the fact that the weights ended up stepping up the total cost to weight curves instead of down to reach local minima points. In the reverse case, if learning rates were chosen to be small enough to accommodate all weight step sizes well enough, the learning process could end up taking a much longer period of time to reach an acceptable level of error.

The Adaptive Moment Estimation (ADAM) Algorithm

In attempt to counteract the flaws of the standard gradient descent algorithm, more advanced methods have since been developed. One such method is Adaptive Moment Estimation (ADAM), which is a process by which network learning and scaling rates are changed adaptively as the learning process of a network progresses [40]. This is done in an effort to set individual and moving learning rates for all of the network weights.

The essential crux of this method is to keep running averages of the gradient function and the squared gradient function in order to scale weight steps as the network continues through its training iterations. The operations that detail these gradient and squared gradient moments can be seen in the equations

$$\mathbf{M}_k = \beta_1 \mathbf{M}_{k-1} + (1 - \beta_1) \nabla J(\mathbf{W})_k \quad \text{(Equation 3.18)}$$

and

$$\mathbf{V}_k = \beta_2 \mathbf{V}_{k-1} + (1 - \beta_2)(\nabla J(\mathbf{W})_k)^2, \quad \text{(Equation 3.19)}$$

where $k = 1,2, \dots, K$ [40]. k represents the current learning iteration index that the network has reached in the training process, K represents the total number of iterations that the network performs the training process for, $\mathbf{M}_k$ is the predicted first order moment of the gradient of the total cost function with respect to the weights of the system at a specific training iteration index, and $\mathbf{V}_k$ is the predicted second order moment of the gradient of the total cost function with respect to the weights of the system at a specific training iteration index. Moreover, β_1 and β_2 are scalar gain values used to control the rate at which the first and second order moments decay or shrink. The scalar decay gain values for the first order and second order moments are respectively 0.9 and 0.999 by default when using the ADAM method. In the case of the first iteration of the training process, the values of the first and second order moments, $\mathbf{M}_0$ and $\mathbf{V}_0$, are set to contain matrices of all zeros.

Because of this starting position, the resulting moment matrices tend to have a zero value bias. To account for the zero moment bias, the moment matrices are scaled depending on the training iteration index of the network learning process. The bias handling expressions for the first and second order moments are

$$\widehat{\mathbf{M}}_k = \frac{\mathbf{M}_k}{1 - \beta_1^k} \quad \text{(Equation 3.20)}$$

and

$$\widehat{\mathbf{V}}_k = \frac{\mathbf{V}_k}{1 - \beta_2^k}, \quad \text{(Equation 3.21)}$$

respectively [40]. In Equation 3.20 and Equation 3.21, $\widehat{\mathbf{M}}_k$ and $\widehat{\mathbf{V}}_k$ are the resulting scaled first and second order moment matrices per iteration of the neural network training process. The equations show the process by which the moment matrices are scaled based on the shrink rate values. The shrink rate values are scaled per training iteration by taking the rate value to the power of the current training iteration index.

After the scaled first and second order moment matrices are found, the final weight stepping expression can be realized. The expression that represents the manipulation of the network weight matrix is [40]

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \frac{\alpha \widehat{\mathbf{M}}_k}{\sqrt{\widehat{\mathbf{V}}_k} + \varepsilon}. \qquad \text{(Equation 3.22)}$$

The only new information in Equation 3.22 is ε, which is a preventative measure incorporated into the equation in order to prevent a possible division by zero situation. As a result, ε is set to some small value as to not affect the process of the weight stepping manipulation. In the default case of the ADAM method, this value is 10^{-8}. Once the process by which a neural network learns has been detailed extensively, the concept of a trained neural network is then evaluated.

3.1.3 Network Configurations, Types of Trained Networks, and Data Models

While the training algorithm of the usage of forward and backward propagation in tandem is certainly important and functions as a sort of metaphorical backbone to the theory of neural networking, without a definite network configuration and end goal in mind, the predictions output by a network do not mean much. This is why the concepts of network configurations and prediction types and models are important. Instead of looking deep into the lower level portions of the learning process, which involve the implementations of the complex mathematics driving the core calculations of a neural network, a higher level approach will be taken in order to adequately describe the aforementioned network configurations and prediction types.

Higher Level Learning Structures

To begin, neural networks usually fall into two general types of learning structures. Those two structures are either unsupervised learning or supervised learning. For the sake of completion, unsupervised learning will be briefly discussed, but it was not the learning structure that ended up being used in this research.

The driving force behind the concept of unsupervised learning is that a network is not given any training output information for determining its predictions [41]. Instead of working like a complex function interpolation tool, a network that is designed for an unsupervised learning purpose is given some input information and makes predictions and changes in any way that it sees fit to do so. To give an example for better context, data clustering is a type of unsupervised learning process. A simple example of how data clustering is used as an unsupervised learning process can be seen in Figure 3.6 [42].

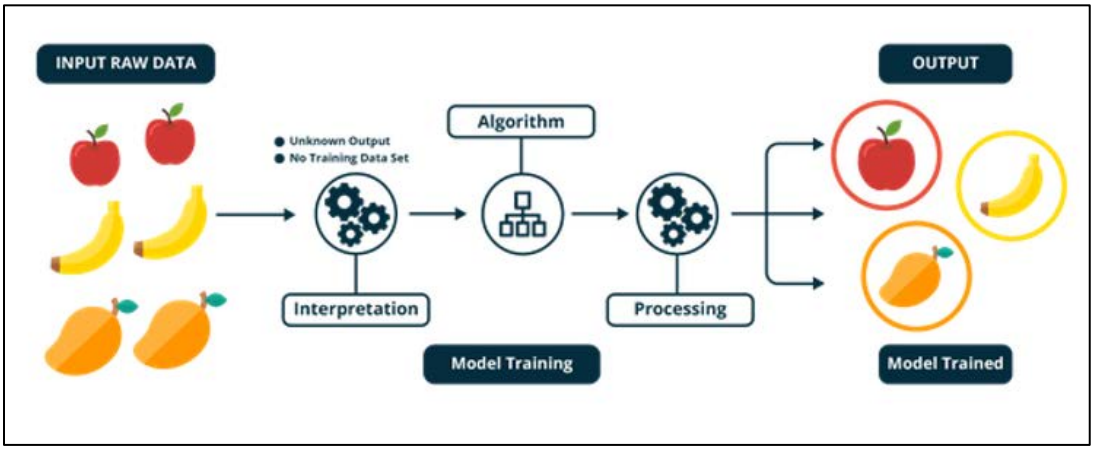

Figure 3.6 Example of clustering as an unsupervised learning process.

With the explanation of unsupervised learning addressed, the concept of supervised learning will now be discussed. Supervised learning was the main learning type that was used during the course of this research. To reiterate the definition of supervised learning, it is the type of learning that takes place when a network or machine learning process is given an example output of what it is trying to predict to consequently learn [43]. The supervised learning technique can be seen as a higher level explanation of the previously described forward and backward propagation process of

a neural network. The overarching idea of supervised learning can also be divided further into smaller concepts. These smaller concepts are known as regression-based learning and classification-based learning. The implementation of these subtopics of supervised learning are dictated by the type of data that is being used for training a network.

Regression

The regression-based learning type is implemented when the data that is used to train networks or machine learning processes is continuous in nature [43]. Since the training data used to train the network is continuous in the case of regression, the prediction the network calculates will also be continuous. This creates a scenario in which a network is treated like a complex, non-linear, interpolating function that that tries to predict a singular output value through its calculations. In theory, the regression-based learning method could be used to predict any value of an output given some input data. However, in practice, predictions of continuous function ranges are often difficult to compensate. This is because continuous values between both input and output variables can vary widely in terms of their scope and features.

To counteract the aforementioned issues, many regression algorithms implement data scaling measures in order to level the information that a network uses for training and predicting. One such algorithm calculates the minimum and maximum values of a range of data and uses that information to scale variables to a new range. This operation is expressed as [44]

$$\mathbf{x}_{\text{new}} = \frac{(\mathbf{x}_{\text{old}} - x_{omin})(x_{nmax} - x_{nmin})}{(x_{omax} - x_{omin})} + x_{nmin}. \qquad \text{(Equation 3.23)}$$

As shown in Equation 3.23, $\mathbf{x}_{\text{new}}$ is a vector containing the newly scaled data of a given variable. This scaling process is done by first finding the absolute minimum value in the original vector of data, represented by x_{omin}, and subtracting that value from every element in the original data vector, represented by $\mathbf{x}_{\text{old}}$. Then, the original vector of data is multiplied and divided by the new and old ranges of the data being scaled, respectively. The original data range is calculated by subtracting the absolute minimum value of the original data from the absolute maximum value of the original data, represented by x_{omax}. The new data range is calculated in the same way.

However, the new minimum and maximum values of the range are chosen rather than found. The values of the absolute minimum and maximum points of the new scaled data vector are represented by x_{nmin} and x_{nmax}, respectively. These new range values can be chosen as anything, but in most cases, the minimum and maximum values are set to zero and one, respectively. This can cause issues in algorithms that calculate the absolute percent error between regression predicted values, so in those cases, other data scaling ranges are used instead to avoid division by zero calculation errors. To complete the scaling algorithm, the minimum of the new data range is added to every element of the now changed old data vector to transform it into the newly scaled data vector. One final process to make note of for regression-based predictions is that if a network prediction is scaled when exiting the network, it must be rescaled back to its original data range in order to preserve the meaning of the output being predicted. This can be achieved by performing the process detailed in Equation 3.23 in reverse.

Classification

The classification-based learning type is implemented when the data that is used to train networks or machine learning processes is discrete in nature [43]. Since the training data used to train the network is discrete in the case of classification, the prediction the network calculates will also be discrete. This creates a scenario in which a network is treated like a state or category estimator that tries to predict the given output through the use of multiple output nodes that are treated as the individual states an output can occupy at any given time. The goal of the network then becomes to activate the output node that represents the correct output category without activating any of the other nodes in the output layer. This idea is illustrated in Figure 3.7 [45].

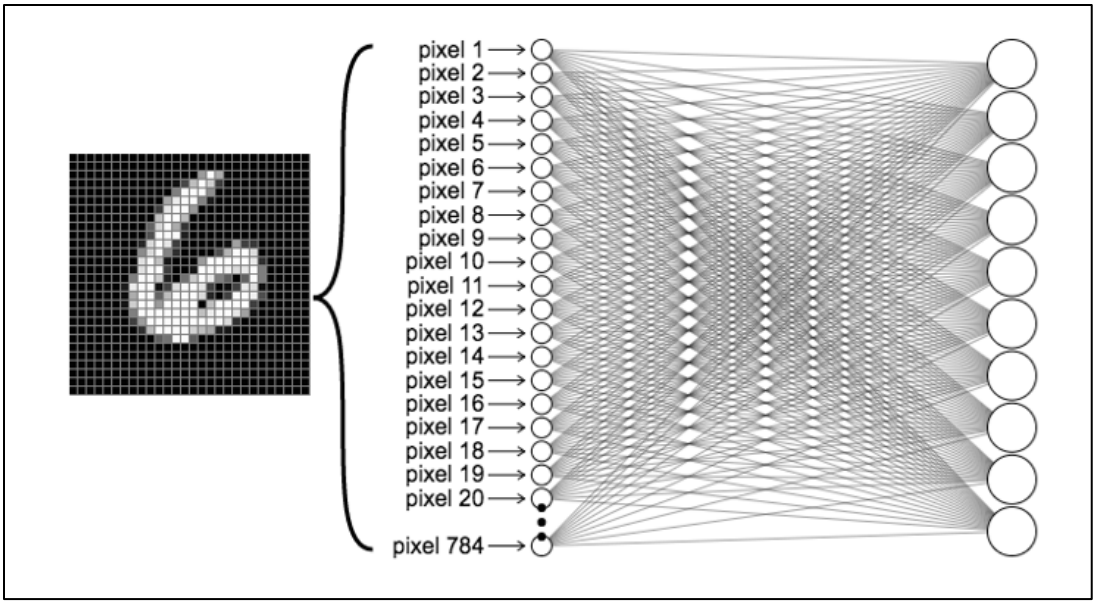

Figure 3.7 Example of classification networks in pixel detection.

While the simplified prediction method that classification networks use is generally well off in terms of making accurate output predictions, this structure comes with the caveat of requiring discrete datasets [46]. This stipulation can at times provide challenges of its own as most data retrieved from sensory measurement devices is continuous in nature. In the case of this research, as well as in other classification application cases, the input data is required to be categorized. While methods exist for categorizing continuous data algorithmically, in this research, all of the data was evaluated, and manual data category enumerations were made in order to utilize the classification learning structure.

In a lot of cases, networks that use the classification learning structure have a comparably easier time of making correct output predictions than networks that use the regression learning structure [43]. This is because the discrete datasets used for classification only have a finite amount of predictable output states, and input data can be more easily generalized into the available states. In contrast, continuous datasets have infinite variability in the values that they can produce. This concept makes output predictions harder to consider accurate without tolerance definitions. In an effort to test multiple prediction techniques, both of these subtopics of supervised learning were used to make predictions.

Network Configurations

Diving deeper into the theory behind network configurations, this idea is what helps scope and build the actual framework into which neural networks are formed. This building process usually starts at the input layer of the network. The choice of inputs and the amount of inputs used in a neural network are prime first subjects for building optimal neural network configurations. This is usually where some outside guidance from an expert is required in picking the right inputs for the type of prediction that is to be made. A common practice in seeing how potential network inputs or variables relate to each other is by performing a cross-correlation evaluation between the potential inputs and the output that is trying to be predicted. The actual formula for the cross-correlation process is [47]

$$r = \frac{\sum_{i=1}^{n}\big((x_i - mx)(y_i - my)\big)}{\sqrt{\sum_{i=1}^{n}(x_i - mx)^2}\sqrt{\sum_{i=1}^{n}(y_i - my)^2}}. \qquad \text{(Equation 3.24)}$$

In the cross-correlation expression, r represents a value on the range of negative one to one. This value is a quantified association measurement between two different system variables. To provide an example case for clarity, if r is positive, there exists a general direct trend in the slopes of the data when comparing both variables. If r is negative, there exists a general inverse trend in the slopes of the data when comparing both variables. When performing a cross-correlation between two system variables, the data corresponding to both variables can be vectorized as **x** and **y**. As such, x_i and y_i would then represent individual data elements being pulled from each respective data vector. Also, mx and my are the respective mean values for each data vector. After the performance of the cross-correlation algorithm on each input with respect to the chosen system output, the input variables can be chosen to reflect the neuron in the input layer of the neural network. In the case of this thesis, variables that had relatively high absolute correlation values with respect to the system output were chosen as inputs to be used in the neural network.

The next portion of putting together neural network configurations is seeing the number of hidden layers and the number of neurons in each hidden layer that should be used to maximize network efficiency and accuracy. To first cover the number of hidden layers to be used, this concept varies depending on what is ultimately desired most from a neural network. Generally, adding more

hidden layers to a neural network increases the ability of the network to pull out specific data features and patterns [48]. These concepts of data feature and pattern extraction through the use of multiple network layers are illustrated graphically in Figure 3.8 [49].

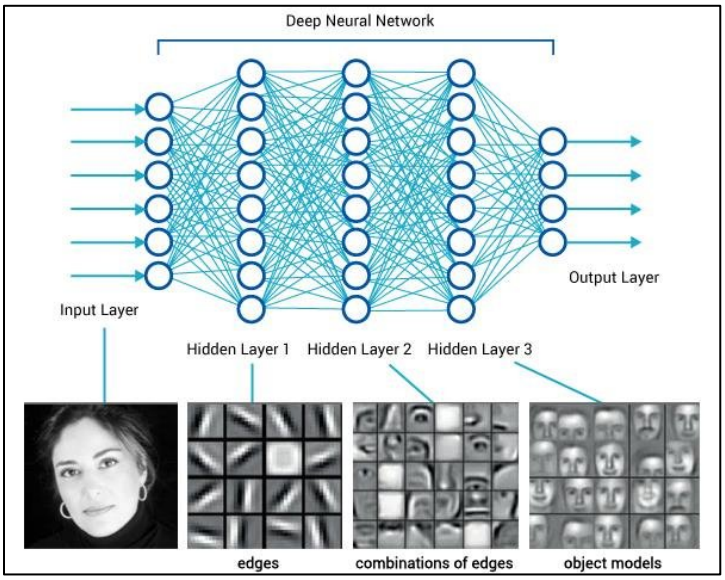

Figure 3.8 Example of feature extraction in a deep learning network.

This multilayer extraction process is known as deep learning due to a network having the ability to learn beyond surface level data, and the network structure itself being deeper than just one or two layers [48]. For usage in this research, neural networks were built using three hidden layers to allow for more complex feature extraction. Going back to an optimized number of neurons in each hidden layer, this selection process is less concrete. A common guideline suggests selecting the number of neurons in the hidden layers to be between the number of input neurons and number of output neurons of the given network [48]. While this guideline provides a range of neurons to start from, this process ultimately comes down to trial and error. In the networks built for this research, the number of neurons selected for each hidden layer was chosen to be half of the number of input neurons. This value seemed to provide favorable network configurations as a whole.

The last portion of the network configuration or architecture that has yet to be fully discussed is the output layer. The output layer of a neural network is usually defined by the type of learning process chosen for the network. If the process is regression-based, the output layer of the given network will usually contain one neuron for each output used to train the network [43]. In the case of this research, only one output was used to train the regression-based networks, and it was decided by steel expert knowledge and usefulness in generalizing the overarching problem of the hook index as a whole [1]. If the process is classification-based, the output layer of the given network will contain neurons pertaining to the number of states that the output variable can occupy [43]. In the case of this research, the number of output states is determined to be three, meaning that the output layer of the network would contain three neurons.

Data Models

One final topic to cover when discussing how neural networks are trained and utilized is about the data models that are used to prepare them for output prediction processes. When choosing a data model to train a network, it is best suited that the selected model has an abundant amount of data. Ideally, this data should detail a variety of different states and conditions that the variables recorded within the data occupy [50]. This is because a neural network has a much easier time predicting a condition that it has already come across in training previously or at least close to a condition that it has encountered in training. In addition, data models that contain the absolute minimum and maximum values a variable could ever achieve are almost always necessities. This is because values outside of this absolute range no longer adhere to the pseudo high order function range that a network has trained to have. When this occurs, output predictions will tend to be inaccurate for any input data point falling outside of the defined network function range.

Another important facet about the use of data models in training neural networks is how the data is divided to accommodate for the purposes of training and testing the network. As a general rule of thumb, a large amount of data is required to train a network. As such, it is good practice to use eighty percent of the data available for training a network and the remaining twenty percent to test how well the trained network will function on data it has not yet encountered [51]. In the classification network case, the remaining twenty percent test data was then sampled to obtain equal amounts of each hook index state for use in normalizing the accuracy rate of each network

that was built. This structure is used to prove that a trained network will be applicable to data that it has not specifically seen before.

Often times, networks are not only tested on data that they have not seen before but also on data that they have seen before. This testing schema will pull a portion of the data that a network has already trained on and test it as well to make sure the network will be useful in many different situations [51]. The training dataset, also known as the validation set, was also sampled to evenly obtain equal amounts of each output state in the terms of a classification network. This resulted in another normalized accuracy rate for each classification network problem.

3.1.4 The Limitations of Neural Networking

Neural networks are an incredibly powerful tool for predicting outcomes when only certain types of data and information are provided. They have their uses in many different applications that require the use of complex data feature extractions and precise amounts of output accuracy. However, neural networks do have limitations and shortcomings. In terms of this research, some limitations are the fact that they are usually far more computationally expensive than other, more simplistic algorithms, they require large datasets in order to make accurate predictions, and they have a black box nature that prevents the understanding of the underlying process that gets used when they make predictions [52].

To cover the most minor shortcoming more thoroughly, for the purposes of this thesis, the neural networks that were built did have some time-based issues. To be more specific, all of the neural networks built for this project required large amounts of time to train effectively. The amount of time required to train the three layer networks that were implemented could be excessively long, depending on the number of input variables and the supervised learning processes that were chosen for the network. These long training periods worked directly against the implementation and testing of different network configurations, as to see whether a network change was beneficial, so a significant amount of time had to be spent letting networks train for comparisons. To briefly compare the time-based performances of the neural networks that were built to the much simpler Naïve Bayesian networks that were built, the Naïve Bayesian networks trained and tested input data much faster than the aforementioned neural network configurations. As stated previously, this

is only a minor shortcoming, as when the neural networks were finished training, output predictions were able to be made at fast speeds with no issues.

The next, more substantial shortcoming that neural networks experienced during the duration of the project was the lack of diverse data. While a dataset was provided by ArcelorMittal for network development, the dataset was almost entirely comprised of data that was not diverse enough to pull meaningful distinctions for outlier hook index cases [1]. This means that most of the data points provided coincided with the most optimal hook index state and neglected the outlier states. Naturally, the data shortcoming issue implies the fact that most of the networks had difficulties differentiating between data points enough to make accurate predictions about the hook index cases. This fact ultimately led to networks that could predict the average hook index cases marginally well but struggled to predict the problem outlier cases. As another direct comparison to the Naïve Bayesian networks, the Naïve networks were able to make more accurate predictions about the output states than the neural network counterparts.

The final, most substantial shortcoming that neural networks experienced was the inability to have a direct understanding of the underlying prediction processes that the networks had made. This is also referred to as the black box problem [52]. This shortcoming ended up affecting the neural networks the most because a future implementation idea of the project is to be able to predict when a poor hook index value or state will happen and effectively implement countermeasures to prevent the outlier hook index cases from occurring. This is simply something that cannot be done with any of the current neural network configuration models. Once again, the Naïve Bayesian networks outperform the neural networks because ultimately, the Naïve networks can draw statistical causality between the inputs that a network accepts and the output prediction that it produces [53]. This shortcoming alone is what spurred the research towards a Bayesian statistics approach, as the algorithms are simpler to calculate, not as much varied data is required to make accurate output predictions, and statistical causality can be drawn from network inputs to validate why a certain output prediction occurred.

3.2 Bayesian Networking

While not directly comparable to some biological process, Bayesian networks are still powerful in terms of prediction capabilities. These networks work as large scale implementations of Bayesian probability theory and statistics. This is what yields these processes quite a few strengths in the context of this research. The most prominent of these strengths is the ability for a Bayesian network to use evidence-based reasoning to provide quantifiable values from which to draw and update predictions [53]. Other benefits of these types of networks are that often times they have the ability to work with smaller, less diverse datasets, and they are generally more computationally efficient than some other machine learning techniques [54]. Keeping these concepts and ideas in mind will be useful for understanding the coming sections when discussing certain aspects of Bayesian networking theory. The coverage of the upcoming sections will include the topics of Naïve Bayesian networks, data sampling methods, the brute force testing method, and the survival testing method.

3.2.1 Naïve Bayesian Networks

To firstly give some background information on the theory of Naïve Bayesian networks, the two major concepts on which the theory of the networks is based are the concepts of conditional probabilities and statistical independence between all input or attribute network random variables [55]. Using these concepts, a network structure can be formed by providing attribute variables from which to draw output or class variable predictions. These predictions are found by inputting data into a system of conditional probability equations that calculate the evidence-based chance of a class variable state occurring given the input information with which a network has been provided. The actual structure of a formed network is arranged in such a way to depict that all of the input or attribute variables are statistically independent from each other and thus the output or class variable is dependent on all of the inputs variables provided. For visual clarity, a graphical illustration of the network structure is viewable in Figure 3.9 [56].

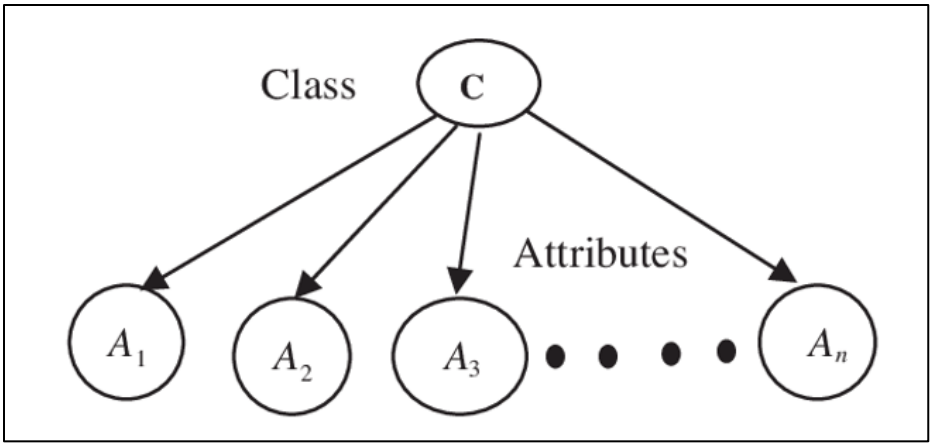

Figure 3.9 Structure of a Naïve Bayesian network.

When all of the attribute data is run through the system of conditional probabilities, the previously input data has now been transformed into a prediction of a class variable state that the network was set to try to predict.

Description of Variable Dependency Types

To understand how and why the random variables of a Naïve Bayesian network are arranged, the ideas of statistical dependency and independency must first be described. First, both statistical dependence and independence are not the same as deterministic dependence and independence. The statistical approach to dependency types is rooted in the fact that data that is being evaluated can have some level of correspondence between other data that is also being evaluated [57]. A good example of this is the description that was given for cross-correlations between variables addressed earlier in this thesis. Variables can have a quantifiable level of correlation regardless of being physically related. In contrast, deterministic dependency types are required to be verifiably intertwined in some physical or scientific process. Data that has a deterministic dependency type will also share the statistical dependency type of the same nature. However, the reverse case of this situation is not always true, as some statistical correlations are purely coincidental.

The actual descriptions of both statistical dependence types are rather straight forward if the thought process mentioned in the previous paragraph is followed. Statistical dependence can be explained as a relation that some number of variables or datasets share in that, when one aspect of

a variable or dataset changes, the other sects of data that are statistically dependent on the variable or dataset that changed will also be changed in some traceable way [57, 58]. Statistical dependence can be directly related to the correlation value that two variables share. If the correlation value between the two variables is relatively larger or smaller than zero, an argument can be made for the statistical dependence of the two variables being evaluated [47].

In a similar fashion, statistical independence can be explained as a relation that some number of variables or datasets share in that when one aspect of a variable or dataset changes, the other sects of data that are statistically independent on the variable or dataset that changed will not be changed in some traceable way. Statistical independence can also be directly related to the correlation value that two variables share. If the correlation value between the two variables is close to zero, an argument can be made for the statistical independence of the two variables being evaluated [47].

Relating these concepts back to the Naïve Bayesian networks that were constructed for this research, all of the attributes in the dataset provided by ArcelorMittal were evaluated to see how they correlated to the class variable. In some network building methods, the highest absolute value correlations were used to pick variables that were shown to have strong statistical dependence on the output variable being tested. When these and all of the other Naïve Bayesian networks were tested, the statistical independence of the input variables chosen was assumed, as this is one of the main conditions to use when implementing Naïve Bayesian classifiers.

Categorical Variable Distributions

Since the Naïve Bayesian networks implemented used classification-based prediction models, the dataset with which they were provided needed to be categorical or discrete in nature [46]. The process by which all of the network variables were categorized was handled manually. This was done by viewing how the data of each variable was distributed and by creating enumerated subdivisions for each variable in the dataset.

The discretization or categorization of the variables used in the Naïve Bayesian networks allows the first major process in using these kinds of networks to take place. This is the process of probability assignments to the variables used in building the networks. This probability assignment

process is handled by what is known as a probability density function. A probability density function essentially pulls information about the distribution of data that represents a certain random variable and assigns probabilities to each individual category or state that a variable can take [59]. This probability assignment process is based on the likelihood of appearance of the states or categories a variable can occupy in the dataset provided.

Different data distributions require different probability density functions. Since the dataset that was used for Bayesian networking was previously categorized, the categorical probability density function was used in the assignment process of variable category probabilities. The equation of the categorical probability density function is

$$P(x_i = t \mid y = c; \alpha) = \frac{N_{tic} + \alpha}{N_c + \alpha n_i}, \qquad \text{(Equation 3.25)}$$

where $i = 1,2, \dots, m$ [60]. To explain more thoroughly, $P(x_i = t \mid y = c; \alpha)$ is a symbolic representation of the probability of x_i, which is an attribute in the given dataset, being equal to t, which is a category of the attribute being evaluated, given that y, which is the class variable of the network, is equal to c, which is a category the class variable can occupy, while using α, which is the Laplacian smoothing value used to account for datasets that do not contain certain attribute or class categories. The value of Laplacian smoothing was set to one as a default value for building networks. N_{tic} is the number of times the category being evaluated appears in the dataset, and N_c is the total number of data samples in the dataset. Then, n_i represents the total number of categories the current attribute has, i is the index of the current attribute being evaluated, and m is the total number of attributes in the network. Through the use of Equation 3.25, all of the categories defined in the network were assigned probabilities for use in network class predictions.

Class Probability Calculations

Using the previously calculated category probabilities from the categorical probability density function, the probabilities of class categories can be calculated in order to choose which class category is more likely given the input data provided to the network. The formula that calculates the class category probabilities given the input data of a network does so by cascading the

probabilities of all the current input variable states together. This operation is shown mathematically as [60]

$$P(y = c \mid x_1 = t, x_2 = t, \dots, x_m = t) = \prod_{i=1}^{m} P(x_i = t \mid y = c; \alpha)\,. \qquad \text{(Equation 3.26)}$$

To detail the operations shown in Equation 3.26, $P(y = c \mid x_1 = t, x_2 = t, \dots, x_m = t)$ represents the probability of the given class variable being equal to the some category, the class variable can be given that all of the system attributes are equal to one of the respective categories that they can be. When this equation is solved using the given data input that was provided to the system for all of the possible class category probabilities, the class category with the highest probability is chosen as the output prediction of the network.

3.2.2 Data Sampling Methods

While the process by which a Naïve Bayesian network makes an output or class variable prediction is relatively simplistic in terms of the computational steps it takes to complete, it is quite powerful in the accuracy of predictions it can make. However, the accuracy of the predictions that are made are still dependent on the initial dataset used to calculate the attribute variable category probabilities. In some cases, like in the case of this research, the dataset provided for attribute category probability calculations is heavily skewed to only have one class category represented. If such a dataset was then used to calculate the different attribute category probabilities, the values received for those probabilities would be intrinsically skewed to favor the most common class category when making predictions with new data that the network had never before encountered. To address this issue, different sampling methods can be used to create a dataset that is more uniform in its distribution of class categories.

Sampling methods are algorithms that review datasets and try to compensate for overrepresented and underrepresented categories present in the data [61]. When implementing a sampling algorithm, a single variable must be chosen for which to determine the category representation bias of the whole dataset. In the case of datasets that are used to train different machine learning

techniques, the variable that is chosen to determine category bias is usually the output or class variable of a network. Once a bias-checking variable has been chosen, the sampling algorithm will add or remove copies of data that are already found in a dataset to make the representation of the categories of the bias checking variable more uniform.

Many different data sampling methods exist to correct for the previously explained dataset issues. Some of them are simplistic in the way that they sample the data to add more samples of different class categories. Others use complex clustering methods to try to include the most valuable information samples relating to the underrepresented or overrepresented class categories [61, 62]. In this research, different data sampling methods were tested, those methods tested being random undersampling, random oversampling, Tomek Links undersampling, ENN undersampling, cluster centroids undersampling, SMOTE oversampling, SMOTE Tomek sampling, and SMOTE ENN sampling, but only one sampling method truly worked well with the given data. That sampling method was the random oversampling method. This data sampling algorithm works by choosing the output or class variable of the dataset to check for category biases in the data [61]. Then, data points pertaining to the underrepresented categories are randomly sampled and placed back into the dataset. This process continues until all of the class categories are represented evenly throughout the whole modified dataset. While the implementation of the random oversampling method did help in increasing the accuracy of the predicted class variable categories, other network methods were implemented to help further increase the accuracy of the Naïve Bayesian networks.

3.2.3 The Brute Force Testing Method

Even with the methods of determining variable correlation values discussed in previous sections of this report, more needed to be done in finding which variables were useful in raising the accuracy of the predictions of the Naïve Bayesian networks. One method that could be implemented that would achieve this purpose is the brute force network building and testing method.

This method was developed for specific usage in this research. What this method aimed to accomplish is the building of every possible network configuration by using every possible combination of input or attribute variables. To prepare the data for use in this algorithm, the

original dataset must be split into randomly assorted portions making up eighty percent and twenty percent of the total dataset. These datasets that were created from the split are called the training and test data, respectively. Then, the training, test, and validation sets of data must be sampled by using the random oversampling method and by choosing the class variable as the way of determining category biases in the data. This is done to give all of the datasets uniform category distributions, so that network accuracy calculations are not heavily skewed in favor of the class variable category that occurs most often. The training dataset will be used to calculate attribute category probabilities, while the test dataset will be used for network accuracy calculations.

To explain the network building section of the algorithm, a network is built following the outputs a binary number counter. A binary counter was used to act as a switch for allowing each available network variable to be used in the network building process. Each bit of the counter represented an input or attribute variable in the dataset. The counter would count decimally from one to two to the power of however many inputs were in the dataset. A mathematical expression of how many networks would need to be created to exhaust all possible input combinations is

$$m = 2^n. \qquad \text{(Equation 3.27)}$$

Where, m is the total amount of networks that need to be built in order to try every network input combination and n represents the total number of inputs included in the dataset. As the decimal counting process progressed, each variable would either be included or not included in the network being built at the current algorithm iteration.

After each network is built, the accuracy values it calculated from the predictions that it made will be compared to the previous network that was built to see if the accuracy of the current network in the algorithm was better than the last. If the accuracy of the current network is better than the accuracy of the previous network, both the accuracy value of the current network and the attributes of the current network will be saved for comparisons against the networks that will be built as the algorithm progresses.

Upon reaching the end of the algorithm, all network combinations are exhausted, and the best possible network configuration is therefore found for the given dataset that was used. Of course, there is one glaring flaw to this algorithm. The amount of networks that need to be built grows exponentially as more and more data attributes are included in a dataset. For smaller datasets, this algorithm could complete without any issues. However, for this algorithm to completely exhaust all possible network configurations while using the dataset that was provided for this research, 2^{99}, or roughly six hundred and thirty-three octillion Naïve Bayesian networks would need to be built and tested. Clearly, this would take far too long to ever complete on any computer, so another algorithm had to be created in order to effectively test different network configurations. This new algorithm also had to abide by the added stipulation of keeping network sizes small enough to use for causal reasoning purposes and to minimize calculation times.

3.2.4 The Survival Testing Method

In an attempt to alleviate the previously discussed issues with the brute force testing method, the survival testing method was created. The survival method can be thought of as a modified brute force method, except that the amount of attribute variables that a given Naïve Bayesian network uses is much smaller than the number of total attribute variables in the dataset used to train the Naïve Bayesian networks. This is due to of the way the survival method operates. The method acts as a variable reduction algorithm that reduces the total number of attributes in a network. The algorithm works on a survival of the fittest mentality in which less useful input variables are cut from the data in order to reduce network sizes and produce better network accuracy results.

To begin using the survival testing algorithm, a dataset containing all of the network attributes and one class variable used for network prediction purposes must be provided. The original dataset must be split into randomly assorted portions making up eighty percent and twenty percent of the total dataset. These datasets that were created from the split are called the training dataset and the test dataset, respectively. After this split of the original dataset occurs, the training dataset must have twenty percent of its contents randomly sampled out from it and copied to a new dataset. This new data created from the training data is known as the validation dataset. Then, the training, test, and validation sets of data must be sampled by using the random oversampling method and by choosing the class variable as the way of determining category biases in the datasets. This is done

to give all of the datasets uniform category distributions so that network accuracy calculations are not heavily skewed in favor of the class variable category that occurs most often. The training data is used to calculate attribute category probabilities, while the validation and test datasets are used for network accuracy calculations.

To start the network building process of the algorithm, the attributes from all of the datasets must be randomized and placed into group sizes of a certain number. For use in this research, the input variables were split into groups of ten. Once the input variables were split into randomized groups, the groups were all treated like new datasets, and the brute force method was used to test every attribute combination of network possible from each of the groupings. As each of the different attribute groupings went through and exhausted the combinations of networks, accuracy calculations were being made. In contrast to the brute force method, accuracy calculations were run on both the test data and validation data in order to see how the networks were performing on data that they both had and had not encountered. These accuracy values were then average together to yield accuracy results that do not favor one dataset over the other. After this, the network configuration that produced the best averaged accuracy calculations had its accuracy values and attributes saved.

After each attribute grouping completely exhausts all of its Naïve Bayesian network configurations, the top three network configurations and average accuracy scores are recorded into data for later comparisons, and any attribute that does not appear in the best network configuration of an attribute grouping is deleted from the training, test, and validation data. After this point in the algorithm, the attributes are rerandomized into groupings, and this process continues to whittle down the available attribute variables that can be used in potential network configurations. The algorithm continues until only a certain number of attributes remains. In relation to this project, the algorithm was chosen to stop once fifteen attributes remained in the data. Once this attribute threshold is reached, all of the remaining attributes are used to build all possible network configurations using the brute force method. As the process takes place, the averaged dataset accuracies are calculated. Once all possible network configurations are exhausted, the top three network configurations are compared to the running top three configurations up to this point, and a final top three best network list is created from this comparison.

As previously stated, the completion of the survival algorithm yields three reduced network structures that perform relatively well. Every time the algorithm was run, it created networks that performed much better than a Naïve Bayesian network that was built using all of the attributes available in the provided dataset. However, this algorithm has shortcomings of its own. For one, it runs off the assumption that the attribute groupings that are formed as the method runs are sufficiently large enough to show whether or not a variable is useful to the structure of a network configuration. As stated before, in the case of this research, the variable grouping size was chosen to be include ten input variables for network calculations, and the threshold for the algorithm to stop at was chosen to be fifteen variables. If these numbers were increased, the algorithm could perform better, but it would take longer to finish. As it was explained in the brute force method section, this is because the number of network configurations that need to be tried increases exponentially as more potential attributes are allowed for network calculations.

Another shortcoming of the survival method is that sometimes groupings of attributes may not always contain the predefined amount of attributes that the program was originally set up for. This is because as the data loses variables, groupings of predefined sizes cannot always be formed, as the number of variables that remain in the data is not always going to be evenly divisible by the predefined grouping size. This can result in groupings of small sizes to be formed, which then fails that assumption that the network structures remain large enough to determine whether or not the attributes in them are useful.

4. RESULTS

Now that the all of the underlying methods that were used to formulate neural networks and Bayesian networks have been explained thoroughly, the results that were received from the networking processes will be presented. This chapter will detail the data that was received from the implementations of the machine learning processes. To aid in the visualization and understanding of this data, the machine learning process results will be displayed tabularly. The presentation of these findings will be expressed in three parts. The first section documents the process behind how network input variables were selected for use in either neural networking or Bayesian networking applications. The second section presents all of the results that were received from the implementation of the neural networking processes. The third section covers all of the results that were received from the implementation of the Bayesian networking processes. An explanation of all of the results will be given in Chapter 5 of the report.

4.1 Data Variable Selection

Since more than one machine learning tactic was used to acquire data and results for this research, there will be two different subsections showing the methods used to determine how inputs were chosen for the network construction of both networking types. The first section is dedicated to neural networks and how their respective data variables were chosen. Similarly, the second section is dedicated to Naïve Bayesian networks and how their attributes and classes were chosen.

4.1.1 Neural Network Variable Selection

The process by which neural network inputs were chosen for usage in the network structures was by the implementation of the cross-correlation function. To be able to implement the cross-correlation algorithm, a network output variable had to first be chosen. After an output variable was chosen, the cross-correlations between the dataset inputs and the output variable could be calculated. To gauge the best overall relation between the inputs and network output, the absolute value of all of the cross-correlation values was taken. The table that depicts the absolute value cross-correlation information that was used in selection of the neural network variables can be seen in Table 4.1.

Table 4.1 Top 20 neural network absolute cross-correlation values.

Neural Network Cross-Correlation Values	
Input Variable Representations	**Relational Value to Output Variable**
1	0.0641
2	0.0607
3	0.0602
4	0.0588
5	0.0572
6	0.0559
7	0.0558
8	0.0557
9	0.0557
10	0.0555
11	0.0552
12	0.0548
13	0.0510
14	0.0510
15	0.0510
16	0.0505
17	0.0502
18	0.0496
19	0.0493
20	0.0490

As the information in Table 4.1 shows, the absolute value cross-correlation values for the top twenty most correlated neural network inputs to the network output were low. This fact is assumed to have negatively affected the performance of the neural networks that were built from these inputs. This configuration of network variables was used in the building process of all of the neural network types that were implemented in this thesis. To draw comparison between the accuracy data received from the neural networks built in this way, all network types were tested while also using all available network inputs.

4.1.2 Naïve Bayesian Network Variable Selection

The process by which Naïve Bayesian network attributes were chosen for usage in the network structures was through the use of two different methods. One was the implementation of the cross-

correlation function, while the other method was through the use of the survival algorithm described in Chapter 3.2.4.

Similar to the neural network application, to be able to implement the cross-correlation algorithm, a network class variable had to be chosen. Moreover, the class variable was included in cross-correlations calculations with every data attribute. Then, the absolute value of each correlational value was taken to determine which attributes were most related to the class. However, this time the data was oversampled with respect to the class categories in order to normalize the representation of each category. The results of these aforementioned calculations can be seen in Table 4.2

Table 4.2 Top 20 Naïve Bayesian network absolute cross-correlation values.

Naïve Bayesian Network Cross-Correlation Values	
Attribute Variable Representations	**Relational Value to Class Variable**
1	0.2673
2	0.2566
3	0.2542
4	0.2255
5	0.2056
6	0.2037
7	0.2026
8	0.2005
9	0.1880
10	0.1861
11	0.1817
12	0.1790
13	0.1737
14	0.1578
15	0.1539
16	0.1534
17	0.1469
18	0.1441
19	0.1428
20	0.1422

As the information in Table 4.2 shows, the absolute value cross-correlation values for the top twenty most correlated Naïve Bayesian network attributes to the network class were marginally higher than the values calculated for the neural networks shown in Table 4.1. This is due to the fact that the dataset was oversampled before the cross-correlation calculation was done on each of the variables. As such, the cross-correlation calculation was able to catch more variation in the dataset, as the data was no longer biased toward a specific hook index category. This configuration of network variables was used in the building process of one Naïve Bayesian network. For comparison, a network was also built using every available variable in the dataset. In addition, a network was built using the variables received from the completion of the survival testing method. Through the completion of this method, an even further reduced network structure was created using only eleven of the best performing attributes, according to the survival method. Now that the selection process of all of the variables has been covered, the actual accuracy results from the built networks will be presented.

4.2 Neural Networking Results

The results collected from all of the different neural network types will also be presented in two subsections. The first will cover the regression-based prediction algorithm, while the second will cover the classification-based algorithm.

4.2.1 Regression-Based Neural Network Percent Error Results

The presentation of the results for both the standard gradient descent and sigmoid training algorithm and the ADAM and ReLU training algorithm will be broken into two separate headings. The standard gradient descent and sigmoid method is tabulated first, while the ADAM and ReLU method appears second.

The Standard Gradient Descent and Sigmoid Method Percent Error Results

To begin the presentation of the results of the regression neural networks, the standard gradient descent and sigmoid method percent error values of the validation dataset are displayed in Table 4.3.

Table 4.3 Validation percent error results from the standard gradient and sigmoid method.

Standard Gradient and Sigmoid Method Validation Percent Error Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	86.2294	94.2719	61.8924	102.5238
Top 20 Corr. Variable Network	83.3196	89.4863	60.7812	99.6914

Next, the standard gradient descent and sigmoid method percent error values of the test dataset are displayed in Table 4.4.

Table 4.4 Test percent error results from the standard gradient and sigmoid method.

Standard Gradient and Sigmoid Method Test Percent Error Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	142.1127	170.8423	101.2546	154.2413
Top 20 Corr. Variable Network	135.0495	156.9521	98.5547	149.6418

Finally, the standard gradient descent and sigmoid method percent error values between the two datasets were averaged and are displayed in Table 4.5.

Table 4.5 Averaged percent error results from the standard gradient and sigmoid method.

Standard Gradient and Sigmoid Method Averaged Percent Error Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	114.1711	132.5571	81.5735	128.3826
Top 20 Corr. Variable Network	109.1846	123.2192	79.6680	124.6666

Now that all of the results relating to the standard gradient descent and sigmoid method have be displayed, the ADAM and ReLU method percent error results will be covered.

The Adaptive Moment Estimation (ADAM) and ReLU Method Percent Error Results

To continue the presentation of the results of the regression neural networks, the ADAM and ReLU method percent error values of the validation dataset are displayed in Table 4.6.

Table 4.6 Validation percent error results from the ADAM and ReLU method.

ADAM and ReLU Method Validation Percent Error Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	84.1472	88.7164	63.5276	100.1975
Top 20 Corr. Variable Network	80.4378	85.2198	58.6347	97.4589

Next, the ADAM and ReLU method percent error values of the test dataset are displayed in Table 4.7.

Table 4.7 Test percent error results from the ADAM and ReLU method.

ADAM and ReLU Method Test Percent Error Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	136.8505	159.9487	97.2948	153.3081
Top 20 Corr. Variable Network	133.3109	157.0364	95.7905	147.1058

Finally, the ADAM and ReLU method percent error values between the two datasets were averaged and are displayed in Table 4.8.

Table 4.8 Averaged percent error results from the ADAM and ReLU method.

ADAM and ReLU Method Averaged Percent Error Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	110.4989	124.3326	80.4112	126.7528
Top 20 Corr. Variable Network	106.8744	121.1281	77.2126	122.2824

Now that all of the results relating to the ADAM and ReLU method have be displayed, the results of the classification-based neural networks will be presented.

4.2.2 Classification-Based Neural Network Accuracy Results

The presentation of the results for both the standard gradient descent and sigmoid training algorithm and the ADAM and ReLU training algorithm will be broken into two separate headings. The standard gradient descent and sigmoid method is displayed first. Afterwards, the ADAM and ReLU method is covered.

The Standard Gradient Descent and Sigmoid Method Accuracy Results

To begin the presentation of the results of the classification neural networks, the standard gradient descent and sigmoid method accuracy values of the validation dataset are displayed in Table 4.9.

Table 4.9 Validation accuracy results from the standard gradient and sigmoid method.

Standard Gradient and Sigmoid Method Validation Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	44.6182	25.8634	90.7421	17.2491
Top 20 Corr. Variable Network	43.0708	27.5923	82.1395	19.4807

Next, the standard gradient descent and sigmoid method accuracy values of the test dataset are displayed in Table 4.10.

Table 4.10 Test accuracy results from the standard gradient and sigmoid method.

Standard Gradient and Sigmoid Method Test Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	32.5572	4.2871	91.8123	1.5723
Top 20 Corr. Variable Network	33.4311	7.0648	89.6254	3.6031

Finally, the standard gradient descent and sigmoid method accuracy values between the two datasets were averaged and are displayed in Table 4.11.

Table 4.11 Averaged accuracy results from the standard gradient and sigmoid method.

Standard Gradient and Sigmoid Method Averaged Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	38.5877	15.0753	91.2772	9.4107
Top 20 Corr. Variable Network	38.2510	17.3286	85.8825	11.5419

Now that all of the results relating to the standard gradient descent and sigmoid method have been presented, the ADAM and ReLU method accuracy results will be shown.

The Adaptive Moment Estimation (ADAM) and ReLU Method Accuracy Results

To continue the presentation of the results of the classification neural networks, the ADAM and ReLU method accuracy values of the validation dataset are displayed in Table 4.12.

Table 4.12 Validation accuracy results from the ADAM and ReLU method.

ADAM and ReLU Method Validation Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	46.1005	27.5849	92.6375	18.0792
Top 20 Corr. Variable Network	44.0924	28.4023	83.7102	20.1648

Next, the ADAM and ReLU method accuracy values of the test dataset are displayed in Table 4.13.

Table 4.13 Test accuracy results from the ADAM and ReLU method.

ADAM and ReLU Method Test Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	33.3700	5.4397	92.7061	1.9641
Top 20 Corr. Variable Network	34.4125	8.1059	91.0042	4.1274

Finally, the ADAM and ReLU method accuracy values between the two datasets were averaged and are displayed in Table 4.14.

Table 4.14 Averaged accuracy results from the ADAM and ReLU method.

ADAM and ReLU Method Averaged Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	39.7353	16.5123	92.6718	10.0217
Top 20 Corr. Variable Network	39.2525	18.2541	87.3572	12.1461

Now that all of the results relating to the ADAM and ReLU method have be displayed, the results of the Naïve Bayesian networks will be presented.

4.3 Naïve Bayesian Networking Results

The results collected from all of the different Naïve Bayesian network types will be presented in this section of the report. The section will cover the results of the all variables network, the cross-correlation variables network, and the survival variables network.

To begin the presentation of the results of the classification Naïve Bayesian networks, the accuracy values of the validation dataset are displayed in Table 4.15.

Table 4.15 Validation accuracy results from the Naïve Bayesian network method.

Naïve Bayes Classifier Validation Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	62.1547	51.6209	89.4561	45.3872
Top 20 Corr. Variable Network	64.0164	56.0234	81.0601	54.9657
Survival Variable Network	68.9191	60.7823	59.6504	86.3247

Next, accuracy values of the test dataset are displayed in Table 4.16.

Table 4.16 Test accuracy results from the Naïve Bayesian network method.

Naïve Bayes Classifier Test Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	48.5638	27.3614	92.8501	25.4798
Top 20 Corr. Variable Network	51.1332	37.1861	82.4927	33.7208
Survival Variable Network	60.0933	53.3633	52.5162	74.4005

Finally, the accuracy values between the two datasets were averaged and are displayed in Table 4.17.

Table 4.17 Averaged accuracy results from the Naïve Bayesian network method.

Naïve Bayes Classifier Averaged Accuracy Results				
(all values are percentages)	**All States**	**Negative Hook Index**	**Optimal Hook Index**	**Positive Hook Index**
All Variable Network	55.3593	39.4912	91.1531	35.4335
Top 20 Corr. Variable Network	57.5748	46.6048	81.7764	44.3433
Survival Variable Network	64.5062	57.0728	56.0833	80.3626

All of these results will be explained and discussed in Chapter 5.

5. DISCUSSION AND CONCLUSION

With the coverage of the results being detailed in the previous chapter of this thesis, this chapter aims to draw necessary conclusions on the presentations of the results and provide suggestions on topics to implement for future work on this research subject. With that being said, the first section of this chapter will cover the discussion of the results received from the neural and Naïve Bayesian networking techniques. The second section will detail suggestions for future work and possible solution implementations for the hook index problem. The last section will explain final thoughts about the research that was done for this thesis.

5.1 Discussion of the Results

To segment the discussion of the results to the sake of clarity, the results will be discussed in the order in which they were received. In other words, the neural network implementations will be discussed in one subsection, and the Naïve Bayesian network implementations will be discussed in another.

5.1.1 Discussion of the Neural Network Results

The implementations of the neutral networking algorithms were the beginning phases of seeing if any predictive influence could be drawn from the dataset that was provided for this research. In these first attempts at using neural networks, regression-based learning methods were tested because they were the most idealistic approach to the problem. This is because if predictions could be made correctly, the value of the output variable representing the hook index could be known at all times within an acceptable degree of error. However, as seen in the results listed in Tables 4.4 through 4.9, the regression neural networks were unable to predict the output variable with acceptable levels of accuracy. The results received from this style of network were not accurate for any set of data that was tested on it either. Normally, networks can at least perform well on sets of data that they have encountered before, but even validation results in this case were inaccurate. Table 4.6 and Table 4.9 show that on average, output values calculated by the network for the most common output state, the optimal hook index state, were just under two times greater or less than

the actual output value that was used as a reference, and this was the best performance of the regression networks.

As can be seen in the aforementioned results tables, different variable configurations were tested, and the ADAM gradient descent algorithm paired with the ReLU neuron activation function was used as well. This was done in an attempt to improve the results over the standard gradient descent, sigmoid, and all input variables networks. The idea behind using fewer data input variables that were more correlated to the output was that networks were possibly having hard times discerning features from the sheer amount of data being input into them. While the network accuracy results did increase due to this change, it was not by much. A similar situation can be seen in the use of the ADAM method with ReLU neuron activations. This method was able to better train networks it was implemented in the same amount of training iterations, but ultimately it was only by a small amount. After seeing how regression neural networks performed, classification neural networks were tested on the dataset.

While classification-based neural networks performed marginally better than their regression-based counterparts, many of the same problems that plagued the regression networks still echoed in the classification network implementations. Reviewing the results presented in Tables 4.10 through 4.15, it is shown that the classification networks were actually able to predict the most common state of the output, the optimal hook index state, quite well. However, this is not necessarily indicative of the prediction capabilities of the implemented networks, since the data was heavily skewed in favor of the optimal hook index state. This means that the classification networks ended up simply guessing the most likely output state, and they ultimately did not provide much in the way of outlier state predictions.

As they were implemented with the regression networks, the top twenty most correlated input variables and the ADAM algorithm with the use of ReLU neuron activation were applied in an attempt to increase network prediction scores. Again, both network configuration changes mostly aided in the increasing of the accuracy scores. However, one statistic to mention was the decrease in the accuracy values of the most common output state, and adversely, the increase in accuracy predictions of the outlier output states. An explanation of this can be described by the fact that

input variables gained discernable features in the calculations of the classification networks when the different input variable configuration was chosen. This resulted in some predictions being directed toward outlier output states rather than being toward the most common output state, as now features that helped predict outlier states could be better seen in the data.

While the neural network techniques seemed to underperform in this application, it is not a very surprising development. The most likely factor of the poor performances of the networks is the dataset that was used for training. This data was heavily skewed in favor of the optimal hook index state of the output variable. While the biased nature of the dataset could have been combated by the implementation of a data sampling algorithm, it was ultimately viewed as unimportant due to the fact that the neural networking techniques would not be able to overcome one major flaw. As described earlier in the thesis, the types of neural networks implemented in this research did not have any capabilities in drawing statistical causation between input variable changes and output variable changes. This fact alone is what led to the research of Bayesian networking techniques.

5.1.2 Discussion of the Naïve Bayesian Network Results

The implementations of Naïve Bayesian networking techniques came from a development that was learned about the hook index problem through the use of neural networks. That development was the need to draw causation relationships between network inputs and outputs. Network causation must be knowable because a direct way of controlling the hook index is what is required to be able to ultimately solve the problem.

Through the use of network probability search queries, a topic brought up in the next subsection, this problem can be alleviated when using Bayesian network structures. As a brief aside, search queries were not applied in the network techniques used in this research, but the insight that could be gained from them is invaluable. These search queries went unimplemented because before any information can be gathered from a Bayesian network, it must be making correct predictions a large portion of the time. As seen by the results found in Tables 4.16 through 4.18, even underperforming Naïve Bayesian networks calculated accuracy results better than the best performing classification neural networks. However, their predictions cannot be seen as concrete

information as of yet. This is because even in the best case network scenario, Naïve Bayesian networks did not perform above a good enough accuracy threshold.

The improvement in accuracies that the Naïve Bayesian networks did see cannot be understated. With just the implementation of a random data oversampling method, these simplistic networks were able to outperform all of the neural networking techniques. The importance of the random sampling method can be seen in Table 4.2, as the cross-correlation values between the attributes and the class variable increased quite a bit. Moreover, the survival algorithm was able to select high performing network variables and make a network structure that performed better with less variable information as a whole. Seeing as the Naïve Bayesian networking model is the most simplistic in nature in terms of the Bayesian networking techniques, there theoretically should be significant room for improvement in prediction calculations from here, especially if data sampling methods and the survival algorithm are used to optimize network structures.

5.2 Future Work Suggestions

While the results shown from this research show promise, there is always room for improvement. As such, there are quite a few steps that could be taken in an effort to push this research even further. The tactics that will be discussed are the implementations of more complex Bayesian networking techniques, the creation of a probability query algorithm that can suggest Bayesian network input values to change to prevent a potentially bad hook index steel roll from being made, a more complex or algorithmic way of creating categorical random variables, and the research and implementation of casual neural network techniques.

The lowest hanging fruit out of these future work implementations is the use of a more complex Bayesian network algorithm to improve results. If this were to be implemented, there is a chance that accuracy results from Bayesian networks would be able to reach a suitable threshold to which causal inferences would start to be meaningful for steel line process implementations. Some potential more complex algorithms to implement would be the Tree Bayes and Hill Climbing algorithms [63].

This is being classified as low hanging fruit, as the only thing required to implement this is to use a different Bayesian network technique in tandem with the survival network variable optimization method. In fact, preexisting tools to implement more complex methods exist in Bayesian learning packages available for the R programming language. The package with these tools is known as bnlearn [64]. However, the problem is that the algorithm was implemented through the use of Python code in this research. There are methods of intermingling Python and R scripts, and methods for doing so were briefly explored for use in this thesis, but ultimately, a rewriting of the code in R syntax would most likely be required to get this working. Alternatively, more complex Bayesian networking techniques could be implemented from scratch in whatever language was necessary, but again this would be a time and research intensive task.

The next suggestion to be made is the creation and use of a probability search query algorithm to find input variables to change in order to prevent poor hook index incidents during the steel making process [63]. As discussed before, an algorithm like this would first require that the Bayesian network in question was actually able to predict hook index issues within a determined acceptable accuracy threshold. Also, in order to reduce complexity of the algorithm and the subsequent implementation of the algorithm into the steel making process, the network implementing the theoretical algorithm would need to not have too many input variables or attributes. This is because the inclusion of more variables would make actual algorithm calculations more complex, and it could potentially be too overbearing of a process to control by a steel line operator who would need to change these values manually. Alternatively, if such an algorithm were to go into practice, an electronic control system could be set up in order to automate the input variable control process.

In terms of actually implementing the algorithm itself, a suggestion can be made for it to function similarly to how the brute force algorithm functions, in that it would run off a counter of some kind that assigned a digit to the state of each input variable category. Then, the probabilities of each category would be multiplied together to calculate the probability of the hook index being in a certain state. This process would try every category combination and output the suggested input variable categories that provide the best chance of making steel with acceptable hook index values. Like with the advanced Bayesian networking techniques, the tools for a speedier development of such an algorithm exist in the R package known as bnlearn [64].

Another potential method to increase accuracy results of the research would be to use a more complex means of categorizing the input dataset. Not very much else can be said on this topic other than a K-means clustering algorithm was used early on in an attempt to categorize the data, but the categories received from it ultimately did not perform as well as the manually created categories [62]. However, a suggestion is being made to find a data categorization algorithm because there were a few times over the course of this research in which certain variables had their data recategorized, and the accuracy results of the classification networks would increase each time.

The final suggestion that can be made is to perform research on and try to implement a kind of neural network that is able to derive some kind of causal relationship between its inputs and outputs [65]. The only direction that can be given in this application is the fact that a paper was reviewed during the process of this research stating that a potential casual neural network structure was possible to implement, but this would require more research into the topic for a better understanding of the application use cases of such a network structure.

5.3 Final Thoughts

Over the course of this research, different machine learning techniques were applied to try to predict the hook index a given roll of steel will experience as it is formed in the steel manufacturing process. The first attempt at prediction of the given hook index of a roll of steel was through the use of use of regression-based neural networks. Even when more sophisticated networking techniques were implemented, regression networks did not make themselves out to be a viable choice for this specific application. After this fact was discovered, classification techniques were put into use. While classification neural networks have the potential to be able to predict hook index states, ultimately, they fall short in their abilities due to their black box calculation processes. As such research was done to implement Naïve Bayesian networks, seeing as how these networks types are among the simplest in Bayesian networking theory, the accuracy results received from their processes can be seen as very successful, as they were able to outperform more sophisticated machine learning algorithms. In fact, if more complex Bayesian algorithms were enacted on the dataset provided, the accuracy results would likely become even better. If research was done on this problem in the future, the suggestion of this thesis would be to continue looking into Bayesian

algorithms to aid in the solution of the hook index issues. With that said, future work into the hook index problem shows great promise, and this research is successful because of the results that were provided.

APPENDIX A. NEURAL NETWORK CODE IMPLEMENTATION

The Main MATLAB Code for Testing Networks

```
clear all; clc; close all;

nl=input('Enter the number of desired hidden layers: ')+2;
nnpl=zeros(1,nl);
for i=2:nl-1
   nnpl(i)=input(['Enter the number of desired neurons in Hidden Layer ' num2str(i-1) ': ']);
end

alpha=0.00001;
val=0.2;

addpath(strcat('.', filesep, 'data'));

csvread('G:\\Thesis_new - Copy\\CSV
Files\\hookData_2019_no_tons_slope_NULL_cat2_osamp.csv',1,3,[1,3,432879,101]);
output_data=csvread('G:\\Thesis_new - Copy\\CSV
Files\\hookData_2019_no_tons_slope_NULL_cat2_osamp.csv',1,102,[1,102,432879,108]);
[num,txt,raw]=xlsread('third_Hook3.csv','A1:C13236');

% randomly select val% of the data rows for verification
% all other rows will be used for training
numrows=size(output_data,1);
rows=randperm(numrows);
trainrows=rows(1:floor((1-val)*numrows));
valrows=rows(floor((1-val)*numrows)+1:end);
nbt=size(input_data(trainrows,:),1);
nbv=size(input_data(valrows,:),1);
```

```
maxinnew=1;
mininnew=0;
maxoutnew=1;
minoutnew=0;

%
[scaled_input_data_t,mininnew,mininold,inoldr,innewr]=data_scaling(input_data(trainrows,:),maxinnew,mininnew,0);
%
[scaled_input_data_v]=data_scaling(input_data(valrows,:),maxinnew,mininnew,1,mininold,inoldr,innewr);
%[scaled_output_data_t,minoutnew,minoutold,outoldr,outnewr]=data_scaling(output_data(trainrows,:),maxoutnew,minoutnew,0);

%[scaled_input_data,mininnew,mininold,inoldr,innewr]=data_scaling(input_data,maxinnew,mininnew,0);
%[scaled_input_data_v,mininnew,mininold,inoldr,innewr]=data_scaling(input_data,maxinnew,mininnew,0);
%[scaled_output_data_t,minoutnew,minoutold,outoldr,outnewr]=data_scaling(output_data,maxoutnew,minoutnew,0);

nnpl(1)=size(input_data,2);
nnpl(nl)=size(output_data,2);

% weights
nn=0;
w=randn(sum(nnpl(2:nl)),max(nnpl)+1);
for i=2:nl
   for j=1:nnpl(i)
      w(nn+j,1:nnpl(i-1)+1)=w(nn+j,1:nnpl(i-1)+1)*sqrt(2/(nnpl(i-1)*nnpl(i)));
   end
```

```
   nn=nn+nnpl(i);
end
dw=zeros(size(w));
y=zeros(nl-1,max(nnpl(2:end)));
u=ones(nl,max(nnpl)+1);
def=zeros(size(y));
sumef=0;
l=0;
it=1000;
mean=0;
var=0;
MAE=zeros(1,it+1);

for k=1:it

[MAE,w,mean,var]=error_calc_wtraining_ADAM(scaled_input_data(trainrows),discrete_output
_data(trainrows),minoutold,minoutnew,outoldr,outnewr,y,u,w,dw,def,sumef,nl,nnpl,nb,l,mean,va
r,alpha,it,MAE);
   %[MSE,w]=error_calc_wtraining_scaled(scaled_input_data_t,scaled_output_data_t,y,u,w,dw,
def,sumef,nl,nnpl,nbt,l,alpha,k,MSE);
end

plot(MAE(1:k))
xlabel('Epotches');ylabel('MAPE');

[MAE]=error_calc_wotraining_new(scaled_input_data_v,output_data(valrows,:),minoutold,min
outnew,outoldr,outnewr,y,u,w,nl,nnpl,nbv,k+1,MAE,[raw(1,:);raw(valrows+1,:)]);

disp('Training error');
disp(MAE(:,k));
disp('Validation error');
```

```
disp(MAE(:,k+1));
```

The MATLAB Code for Data Scaling

```
function [scaled_data,minnew,minold,oldr,newr] =
data_scaling(data,maxnew,minnew,typescale,minold,oldr,newr)

scaled_data=zeros(size(data));
if(typescale==0)
   maxold=zeros(1,size(data,2));
   minold=zeros(1,size(data,2));
   oldr=zeros(1,size(data,2));
   newr=maxnew-minnew;
   for i=1:size(data,2)
      maxold(i)=max(data(:,i));
      minold(i)=min(data(:,i));
      oldr(i)=maxold(i)-minold(i);
      scaled_data(:,i)=(((data(:,i)-minold(i))*newr)/oldr(i))+minnew;
   end
else
   for i=1:size(data,2)
      scaled_data(:,i)=(((data(:,i)-minold(i))*newr)/oldr(i))+minnew;
   end
end
```

The MATLAB Code for Training Networks Using the Standard Gradient Descent Method

```
function [MSE,w] =
error_calc_wtraining_scaled(input_data,output_data,y,u,w,dw,def,sumef,nl,nnpl,nb,l,alpha,it,MS
E)
```

```
for i=1:nb
   u(1,2:nnpl(1)+1)=input_data(i,:);
   %[u,nn]=forward_prop_sig(u,y,w,nl,nnpl);
   [u,nn]=forward_prop_sig_lin(u,y,w,nl,nnpl);
   %[u,nn]=forward_prop_ReLU_lin(u,y,w,nl,nnpl);
   %[u,nn]=forward_prop_ReLU(u,y,w,nl,nnpl);
   e=u(nl,2:nnpl(nl)+1)-output_data(i,:);
   %[dw]=back_prop_sig(e,u,w,dw,def,sumef,nl,nnpl,nn,l);
   [dw]=back_prop_sig_lin(e,u,w,dw,def,sumef,nl,nnpl,nn,l);
   %[dw]=back_prop_ReLU_lin(e,u,w,dw,def,sumef,nl,nnpl,nn,l);
   %[dw]=back_prop_ReLU(e,u,w,dw,def,sumef,nl,nnpl,nn,l);
   w=w-(alpha*dw);
   %de=e./output_data(i,:);
   %ape=abs(de)*100;
   %tape=sum(ape);
   %tape=te^2;
   e2=e.^2;
   MSE(:,it)=MSE(:,it)+e2';
end
MSE(:,it)=MSE(:,it)/nb;
```

The MATLAB Code for Training Networks Using the ADAM Gradient Descent Method

```
function [MAE,w,mean,var] =
error_calc_wtraining_ADAM(input_data,output_data,minoutold,minoutnew,outoldr,outnewr,y,u,
w,dw,def,sumef,nl,nnpl,nb,l,mean,var,alpha,it,MAE)

for i=1:nb
   u(1,2:nnpl(1)+1)=input_data(i,:);
   %[u,nn]=forward_prop_sig(u,y,w,nl,nnpl);
   %[u,nn]=forward_prop_sig_lin(u,y,w,nl,nnpl);
```

```
    %[u,nn]=forward_prop_ReLU_lin(u,y,w,nl,nnpl);
    [u,nn]=forward_prop_ReLU(u,y,w,nl,nnpl);
    e=((((u(nl,2:nnpl(nl)+1)-minoutnew).*outoldr)./outnewr)+minoutold)-output_data(i,:);
    %[dw]=back_prop_sig(e,u,w,dw,def,sumef,nl,nnpl,nn,l);
    %[dw]=back_prop_sig_lin(e,u,w,dw,def,sumef,nl,nnpl,nn,l);
    %[dw]=back_prop_ReLU_lin(e,u,w,dw,def,sumef,nl,nnpl,nn,l);
    [dw]=back_prop_ReLU(e,u,w,dw,def,sumef,nl,nnpl,nn,l);
    dw2=dw.^2;
    [mean]=rec_dec_avg(mean,dw,0.9,((it-1)*nb)+i);
    [var]=rec_dec_avg(var,dw2,0.999,((it-1)*nb)+i);
    [dw]=ADAM(mean,var,alpha,1e-8);
    w=w-dw;
    %de=e./output_data(i,:);
    %ape=abs(de)*100;
    %tape=sum(ape);
    %tape=te^2;
    e2=e.^2;
    %ae=abs(e);
    mae=sum(e2)/length(e);
    MAE(:,it)=MAE(:,it)+mae;
end
MAE(:,it)=MAE(:,it)/nb;
MAE=[MAE MAE(:,it)];
```

The MATLAB Code for Performing Forward Propagation Using Sigmoid Activation Functions

```
function [u,nn] = forward_prop_sig(u,y,w,nl,nnpl)

nn=0;
for j=2:nl
```

```
    for k=1:nnpl(j)
        y(j-1,k)=w(nn+k,1:nnpl(j-1)+1)*u(j-1,1:nnpl(j-1)+1)';
    end
    u(j,2:k+1)=1./(1+exp(-y(j-1,1:k)));
    nn=nn+nnpl(j);
end
```

The MATLAB Code for Performing Forward Propagation Using Sigmoid and Linear Activation Functions

```
function [u,nn] = forward_prop_sig_lin(u,y,w,nl,nnpl)

nn=0;
for j=2:nl
    for k=1:nnpl(j)
        y(j-1,k)=w(nn+k,1:nnpl(j-1)+1)*u(j-1,1:nnpl(j-1)+1)';
    end
    if (j~=nl)
        u(j,2:k+1)=1./(1+exp(-y(j-1,1:k)));
    else
        u(j,2:k+1)=y(j-1,1:k);
    end
    nn=nn+nnpl(j);
end
```

The MATLAB Code for Performing Forward Propagation Using ReLU Activation Functions

```
function [u,nn] = forward_prop_ReLU(u,y,w,nl,nnpl)

nn=0;
```

```
for j=2:nl
   for k=1:nnpl(j)
      y(j-1,k)=w(nn+k,1:nnpl(j-1)+1)*u(j-1,1:nnpl(j-1)+1)';
   end
   u(j,2:k+1)=max([zeros(j-1,k); y(j-1,1:k)]);
   nn=nn+nnpl(j);
end
```

The MATLAB Code for Performing Forward Propagation Using ReLU and Linear Activation Functions

```
function [u,nn] = forward_prop_ReLU_lin(u,y,w,nl,nnpl)

nn=0;
for j=2:nl
   for k=1:nnpl(j)
      y(j-1,k)=w(nn+k,1:nnpl(j-1)+1)*u(j-1,1:nnpl(j-1)+1)';
   end
   if (j~=nl)
      u(j,2:k+1)=max([zeros(j-1,k); y(j-1,1:k)]);
   else
      u(j,2:k+1)=y(j-1,1:k);
   end
   nn=nn+nnpl(j);
end
```

The MATLAB Code for Performing Backward Propagation Using Sigmoid Activation Functions

```
function [dw] = back_prop_sig(e,u,w,dw,def,sumef,nl,nnpl,nn,l)
```

```
% def(nl+1,1:no)=(-e./((output_data.^2).*abs(e)));
def(nl-1,1:nnpl(nl))=2*e;
for i=nl-1:-1:2
   for j=nnpl(i):-1:1
      for k=nnpl(i+1):-1:1
         sumef=sumef+(def(i,k)*(u(i+1,k+1)*(1-u(i+1,k+1)))*(w(nn-l,j+1)));
         l=l+1;
      end
      def(i-1,j)=sumef;
      sumef=0;
      l=0;
   end
   nn=nn-nnpl(i+1);
end
nn=0;
for i=2:nl
   dw(nn+1:nn+nnpl(i),1:nnpl(i-1)+1)=(def(i-1,1:nnpl(i)).*u(i,2:nnpl(i)+1).*(1-
u(i,2:nnpl(i)+1)))'*u(i-1,1:nnpl(i-1)+1);
   nn=nn+nnpl(i);
end
```

The MATLAB Code for Performing Backward Propagation Using Sigmoid and Linear Activation Functions

```
function [dw] = back_prop_sig_lin(e,u,w,dw,def,sumef,nl,nnpl,nn,l)

% def(nl+1,1:no)=(-e./((output_data.^2).*abs(e)));
def(nl-1,1:nnpl(nl))=2*e;
for i=nl-1:-1:2
   for j=nnpl(i):-1:1
      for k=nnpl(i+1):-1:1
```

```
            if (i~=nl-1)
               sumef=sumef+(def(i,k)*(u(i+1,k+1)*(1-u(i+1,k+1)))*(w(nn-l,j+1)));
            else
               sumef=sumef+(def(i,k)*(w(nn-l,j+1)));
            end
            l=l+1;
         end
         def(i-1,j)=sumef;
         sumef=0;
         l=0;
      end
      nn=nn-nnpl(i+1);
end
nn=0;
for i=2:nl
   if (i~=nl)
      dw(nn+1:nn+nnpl(i),1:nnpl(i-1)+1)=(def(i-1,1:nnpl(i)).*u(i,2:nnpl(i)+1).*(1-
u(i,2:nnpl(i)+1)))'*u(i-1,1:nnpl(i-1)+1);
   else
      for j=1:nnpl(i)
         dw(nn+j,1:nnpl(i-1)+1)=def(i-1,j)*u(i-1,1:nnpl(i-1)+1);
      end
   end
   nn=nn+nnpl(i);
end
```

The MATLAB Code for Performing Backward Propagation Using ReLU Activation Functions

```
function [dw] = back_prop_ReLU(e,u,w,dw,def,sumef,nl,nnpl,nn,l)
```

```
% def(nl+1,1:no)=(-e./((output_data.^2).*abs(e)));
def(nl-1,1:nnpl(nl))=2*e;
for i=nl-1:-1:2
   for j=nnpl(i):-1:1
      for k=nnpl(i+1):-1:1
         if (u(i+1,k+1)>0)
            sumef=sumef+(def(i,k)*(w(nn-l,j+1)));
         end
         l=l+1;
      end
      def(i-1,j)=sumef;
      sumef=0;
      l=0;
   end
   nn=nn-nnpl(i+1);
end
nn=0;
for i=2:nl
   for j=1:nnpl(i)
      if (u(i,j+1)>0)
         dw(nn+j,1:nnpl(i-1)+1)=def(i-1,j)*u(i-1,1:nnpl(i-1)+1);
      end
   end
   nn=nn+nnpl(i);
end
```

The MATLAB Code for Performing Backward Propagation Using ReLU and Linear Activation Functions

```
function [dw] = back_prop_ReLU_lin(e,u,w,dw,def,sumef,nl,nnpl,nn,l)
```

```
% def(nl+1,1:no)=(-e./((output_data.^2).*abs(e)));
def(nl-1,1:nnpl(nl))=2*e;
for i=nl-1:-1:2
   for j=nnpl(i):-1:1
      for k=nnpl(i+1):-1:1
         if (i~=nl-1)
            if (u(i+1,k+1)>0)
               sumef=sumef+(def(i,k)*(w(nn-l,j+1)));
            end
         else
            sumef=sumef+(def(i,k)*(w(nn-l,j+1)));
         end
         l=l+1;
      end
      def(i-1,j)=sumef;
      sumef=0;
      l=0;
   end
   nn=nn-nnpl(i+1);
end
nn=0;
for i=2:nl
   if (i~=nl)
      for j=1:nnpl(i)
         if (u(i,j+1)>0)
            dw(nn+j,1:nnpl(i-1)+1)=def(i-1,j)*u(i-1,1:nnpl(i-1)+1);
         end
      end
   else
      for j=1:nnpl(i)
         dw(nn+j,1:nnpl(i-1)+1)=def(i-1,j)*u(i-1,1:nnpl(i-1)+1);
```

```
      end
   end
   nn=nn+nnpl(i);
end
```

The MATLAB Code for Calculating the Decaying Gradient Averages in the ADAM Method

```
function [avg] = rec_dec_avg(avg,val,c,it)

avg=(c*avg)+(1-c)*val;
avg=avg/(1-(c^it));
```

The MATLAB Code for Updating the Network Gradient Values in the ADAM Method

```
function [dw] = ADAM(mean,var,alpha,epsilon)

dw=(alpha./((var.^(1/2))+epsilon)).*mean;

end
```

The MATLAB Code for Performing Predictions on Test Data

```
function [MAE] =
error_calc_wotraining_new(input_data,output_data,minoutold,minoutnew,outoldr,outnewr,y,u,w
,nl,nnpl,nb,it,MAE,labels)

varnam={};
for i=1:nnpl(nl)
   varnam=[varnam ['Actual Output ' num2str(i)] ['Predicted Output ' num2str(i)] ['Absolute
Error ' num2str(i)] ['Percent Error ' num2str(i)]];
```

```
end
CSV=zeros(nb,4*nnpl(nl));
for i=1:nb
   u(1,2:nnpl(1)+1)=input_data(i,:);
   [u]=forward_prop_sig(u,y,w,nl,nnpl);
   %[u]=forward_prop_sig_lin(u,y,w,nl,nnpl);
   %[u]=forward_prop_ReLU_lin(u,y,w,nl,nnpl);
   %[u]=forward_prop_ReLU(u,y,w,nl,nnpl);
   e=((((u(nl,2:nnpl(nl)+1)-minoutnew).*outoldr)./outnewr)+minoutold)-output_data(i,:);
   ae=abs(e);
   de=e./output_data(i,:);
   ape=abs(de)*100;
   %tape=sum(ape);
   %tape=te^2;
   %e2=e.^2;
   MAE(:,it)=MAE(:,it)+ae';
   k=1;
   for j=0:4:3*nnpl(nl)
      CSV(i,j+1:j+4)=[output_data(i,k) ((((u(nl,k+1)-
minoutnew)*outoldr(k))/outnewr)+minoutold(k)) ae(k) ape(k)];
      k=k+1;
   end
end
CSV_c=[varnam; num2cell(CSV)];
data=[labels CSV_c];
writetable(cell2table(data),'test.csv','writevariablenames',0);
MAE(:,it)=MAE(:,it)/nb;
```

APPENDIX B. BAYESIAN NETWORK CODE IMPLEMENTATION

The Main Python Code for Testing Individual Networks

```
import functions as func
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import CategoricalNB
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix
from sklearn.metrics import log_loss

dual_data=False

if dual_data==True:
    X_train,y_train=func.data_init(data_file='D:\\R Thesis Files\\test.csv')
    X_test,y_test=func.data_init(data_file='D:\\R Thesis Files\\Jan_feb_2020Data_test.csv')

else:
    X,y=func.data_init(data_file='D:\\R Thesis Files\\csv\\all_data_cat.csv',output='Asym_Cld')
    X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2)



X_train_samp,y_train_samp=func.data_samp(X_train,y_train,samp_type='Random
Oversampling')
X_not,X_val,y_not,y_val=train_test_split(X_train,y_train,test_size=0.2)
X_val_samp,y_val_samp=func.data_samp(X_val,y_val,samp_type='Random Oversampling')
X_test_samp,y_test_samp=func.data_samp(X_test,y_test,samp_type='Random Oversampling')
clf=GaussianNB()
```

```
nb=clf.fit(X_train_samp,y_train_samp)

y_predv=nb.predict(X_val_samp)
y_predvp=nb.predict_proba(X_val_samp)
#y_predv=nb.predict(X_val)
cm_val=confusion_matrix(y_val_samp,y_predv)
lessacc_val,betweenacc_val,greatacc_val,totalacc_val=func.accuracy(cm_val,y_val_samp,y_pre
dv)
logv=log_loss(y_val_samp,y_predvp,eps=1e-15)

y_predt=nb.predict(X_test_samp)
y_predtp=nb.predict_proba(X_test_samp)
#y_predt=nb.predict(X_test)
cm_test=confusion_matrix(y_test_samp,y_predt)
lessacc_test,betweenacc_test,greatacc_test,totalacc_test=func.accuracy(cm_test,y_test_samp,y_p
redt)
logt=log_loss(y_test_samp,y_predtp,eps=1e-15)
```

The Python Code Containing the Functions Used for Data Initialization, Data Sampling, and Accuracy Calculations

```
import pandas as pd
import numpy as np
import imblearn.under_sampling as und_samp
import imblearn.over_sampling as ov_samp
import imblearn.combine as com_samp

from sklearn.metrics import accuracy_score
```

```
def data_init(data_file='G:\\Thesis_new - Copy\\CSV
Files\\test_Asym_Cld.csv',limits_file='G:\\Thesis_new - Copy\\CSV Files\\Category
Limits.csv',output='Asym_Cld',categorize=False,csv_write=False):

    df=pd.read_csv(data_file,header=None,low_memory=False).to_numpy()
    size_df=df.shape
    num_df_rows=size_df[0]
    num_df_columns=size_df[1]
    data=df[1:num_df_rows,3:num_df_columns]
    size_data=data.shape
    num_data_rows=size_data[0]
    num_data_columns=size_data[1]
    num_data_outputs=7
    num_data_inputs=size_data[1]-num_data_outputs

    if categorize==False:
        X=data[:,0:num_data_inputs]
        out=data[:,num_data_inputs:num_data_inputs+num_data_outputs]

    else:
        limits=pd.read_csv(limits_file).to_numpy()
        size_limits=limits.shape
        num_limit_rows=size_limits[0]
        num_limit_columns=size_limits[1]
        num_limit_inputs=size_limits[1]-num_data_outputs

        X=data[:,0].reshape(num_data_rows,1)
        temp=data[:,1:num_data_columns]
        for j in range(num_limit_columns):
            for i in range(num_data_rows):
                if j!=0 and j!=1:
```

```
                x=float(temp[i,j])
            else:
                x=temp[i,j]
            for k in range(num_limit_rows):
                if limits[k][j]!='':
                    if eval(limits[k][j]):
                        if j!=0:
                            temp[i,j]=k + 1
                        else:
                            split=[char for char in x]
                            split[0]=str(k)
                            temp_str=""
                            temp[i,j]=int(temp_str.join(split))
                        break
                    else:
                        break

        cat_inputs=temp[:,0:num_limit_inputs]
        X=np.append(X,cat_inputs,1)
        out=temp[:,num_limit_inputs:num_limit_inputs+num_data_outputs]

    if csv_write==True:
        input_headers=df[0,3:num_data_inputs+3].reshape(1,num_data_inputs)
        X_df=np.append(input_headers,X,0)

output_headers=df[0,num_data_inputs+3:num_data_inputs+num_data_outputs+3].reshape(1,nu
m_data_outputs)
        out_df=np.append(output_headers,out,0)
        CSV_df=df[:,0:3]
        CSV_df=np.append(CSV_df,X_df,1)
        CSV_df=np.append(CSV_df,out_df,1)
```

```
        pd.DataFrame(CSV_df).to_csv(path_or_buf='G:\\Thesis_new - Copy\\CSV
Files\\test.csv',header=False,index=False)

    if output=='Asym_Cld':
        y=out[:,0]

    elif output=='CLD':
        y=out[:,1]

    elif output=='ASYM':
        y=out[:,2]

    elif output=='F7_hook_Index_1':
        y=out[:,3]

    elif output=='F7_hook_Index_2':
        y=out[:,4]

    elif output=='F7_hook_Index_3':
        y=out[:,5]

    elif output=='F7_hook_Index':
        y=out[:,6]

    return X.astype(np.int),y.astype(np.int)
    #return X.astype(np.float),y.astype(np.float)

def data_samp(X,y,samp_type='None'):

    if samp_type=='None':
        X_samp=X
```

```
    y_samp=y

elif samp_type=='Random Undersampling':
    sm=und_samp.RandomUnderSampler(sampling_strategy='majority')
    X_samp,y_samp=sm.fit_resample(X,y)

elif samp_type=='Random Oversampling':
    sm=ov_samp.RandomOverSampler()
    X_samp,y_samp=sm.fit_sample(X,y)

elif samp_type=='Tomek Links Undersampling':
    sm=und_samp.TomekLinks(sampling_strategy='majority')
    X_samp,y_samp=sm.fit_resample(X,y)

elif samp_type=='ENN Undersampling':
    sm=und_samp.EditedNearestNeighbours(sampling_strategy='majority')
    X_samp,y_samp=sm.fit_resample(X,y)

elif samp_type=='Cluster Centroids Undersampling':
    sm=und_samp.ClusterCentroids(sampling_strategy='majority')
    X_samp,y_samp=sm.fit_resample(X,y)

elif samp_type=='SMOTE Oversampling':
    sm=ov_samp.SMOTE(k_neighbors=8)
    X_samp,y_samp=sm.fit_sample(X,y)

elif samp_type=='SMOTE Tomek Sampling':
    sm=com_samp.SMOTETomek()
    X_samp,y_samp=sm.fit_sample(X,y)

elif samp_type=='SMOTE ENN Sampling':
```

```
sm=com_samp.SMOTEENN(enn=und_samp.EditedNearestNeighbours(sampling_strategy='maj
ority',kind_sel='mode'))
        X_samp,y_samp=sm.fit_sample(X,y)

    return X_samp,y_samp

def accuracy(cm,y,y_pred):

    less=cm[0][0]+cm[0][1]+cm[0][2]
    between=cm[1][0]+cm[1][1]+cm[1][2]
    great=cm[2][0]+cm[2][1]+cm[2][2]
    lessacc=cm[0][0]/less
    betweenacc=cm[1][1]/between
    greatacc=cm[2][2]/great
    totalacc=accuracy_score(y,y_pred)

    return lessacc,betweenacc,greatacc,totalacc
```

The Python Code for Using the Brute Force Testing Method

```
import functions as func
import pandas as pd
import numpy as np
import math

from sklearn.naive_bayes import CategoricalNB
from sklearn.metrics import confusion_matrix

X_train,y_train=func.data_init(data_file='D:\\R Thesis Files\\csv\\train10.csv')
X_test,y_test=func.data_init(data_file='D:\\R Thesis Files\\csv\\test10.csv')
#X_train=np.ones([10,3])
```

```
#y_train=np.array([0,1,2,0,1,2,0,1,2,0])
#X_test=np.ones([10,3])
#y_test=np.array([0,0,0,1,1,1,2,2,2,2])
clf=CategoricalNB()

i=1
top3_index=np.zeros([3,X_train.shape[1]])
top3_acc=np.zeros([3,4])
acc_mat=np.zeros([4])
index_mat=np.zeros([X_train.shape[1]])

while i<2**X_train.shape[1]:
    num_bits=int(math.floor(math.log2(i))+1)
    bits=[(i >> bit) & 1 for bit in range(num_bits - 1, -1, -1)]
    bits.reverse()
    result=np.where(np.array(bits) == 1)
    index_mat[0:result[0].shape[0]]=result[0]
    nb=clf.fit(X_train[:,result[0]],y_train)
    y_predt=nb.predict(X_test[:,result[0]])
    cm_test=confusion_matrix(y_test,y_predt)
    acc_mat[1],acc_mat[2],acc_mat[3],acc_mat[0]=func.accuracy(cm_test,y_test,y_predt)
    if acc_mat[0]>=top3_acc[2,0]:
        top3_acc[2,:]=acc_mat
        top3_index[2,:]=index_mat
        for j in range(1,-1,-1):
            if acc_mat[0]>=top3_acc[j,0]:
                top3_acc[j+1,:]=top3_acc[j,:]
                top3_index[j+1,:]=top3_index[j,:]
                top3_acc[j,:]=acc_mat
                top3_index[j,:]=index_mat
            else:
```

```
            break
    i+=1
    index_mat=np.zeros([X_train.shape[1]])

pd.DataFrame(top3_acc).to_csv(path_or_buf='D:\\R Thesis
Files\\top3_acc.csv',header=False,index=False)
pd.DataFrame(top3_index).to_csv(path_or_buf='D:\\R Thesis
Files\\top3_index.csv',header=False,index=False)
```

The Python Code for Using the Survival Testing Method

```
import functions as func
import pandas as pd
import numpy as np
import math
import random

from sklearn.naive_bayes import CategoricalNB
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

X,y=func.data_init(data_file='D:\\R Thesis Files\\csv\\all_data_cat.csv',output='Asym_Cld')
df_labels=pd.read_csv('D:\\R Thesis
Files\\csv\\all_data_cat.csv',header=None,low_memory=False).to_numpy()
labels=np.empty([1,df_labels[0,3:102].shape[0]],'O')
labels[0,:]=df_labels[0,3:102]

X_train_ns,X_test_ns,y_train_ns,y_test_ns=train_test_split(X,y,test_size=0.2)
X_not,X_val_ns,y_not,y_val_ns=train_test_split(X_train_ns,y_train_ns,test_size=0.2)

X_train_nl,y_train=func.data_samp(X_train_ns,y_train_ns,samp_type='Random Oversampling')
```

```
X_val_nl,y_val=func.data_samp(X_val_ns,y_val_ns,samp_type='Random Oversampling')
X_test_nl,y_test=func.data_samp(X_test_ns,y_test_ns,samp_type='Random Oversampling')

clf=CategoricalNB()
net_max_size=10
num_runs=30
top3total_index=np.empty([3,net_max_size+5],'O')
top3total_acc=np.zeros([3,4])

for cnt in range(num_runs):
    top3_index=np.empty([3,net_max_size+5],'O')
    top3_acc=np.zeros([3,4])
    X_train=np.append(labels.astype(np.dtype('O')),X_train_nl.astype(np.dtype('O')),0)
    X_val=np.append(labels.astype(np.dtype('O')),X_val_nl.astype(np.dtype('O')),0)
    X_test=np.append(labels.astype(np.dtype('O')),X_test_nl.astype(np.dtype('O')),0)
    while X_train.shape[1]>net_max_size+5:
        ins=np.array([range(X_train.shape[1])])
        X_train_copy=np.array(X_train)
        X_val_copy=np.array(X_val)
        X_test_copy=np.array(X_test)
        np.random.shuffle(np.transpose(ins))
        num_nets=math.ceil(ins.shape[1]/net_max_size)
        for j in range(num_nets):
            if j!=num_nets-1:
                net_ins=ins[0,j*net_max_size:((j+1)*net_max_size)]
                X_train_temp_labels=X_train_copy[0,net_ins]
                X_train_temp_data=X_train_copy[1:X_train_copy.shape[0],net_ins].astype(np.int)
                X_val_temp_data=X_val_copy[1:X_val_copy.shape[0],net_ins].astype(np.int)
                X_test_temp_data=X_test_copy[1:X_test_copy.shape[0],net_ins].astype(np.int)
                topminor_acc=np.zeros([4])
                topminor_index=np.empty([net_max_size],'O')
```

```
            acc_mat=np.zeros([4])
            index_mat=np.empty([net_max_size],'O')
            i=1
            while i<2**X_train_temp_data.shape[1]:
                num_bits=int(math.floor(math.log2(i))+1)
                bits=[(i >> bit) & 1 for bit in range(num_bits - 1, -1, -1)]
                bits.reverse()
                result=np.where(np.array(bits) == 1)
                index_mat[result[0]]=X_train_temp_labels[result[0]]
                nb=clf.fit(X_train_temp_data[:,result[0]],y_train)
                y_predv=nb.predict(X_val_temp_data[:,result[0]])
                y_predt=nb.predict(X_test_temp_data[:,result[0]])
                cm_val=confusion_matrix(y_val,y_predv)
                cm_test=confusion_matrix(y_test,y_predt)

lessacc_val,betweenacc_val,greatacc_val,totalacc_val=func.accuracy(cm_val,y_val,y_predv)

lessacc_test,betweenacc_test,greatacc_test,totalacc_test=func.accuracy(cm_test,y_test,y_predt)

#acc_mat[1],acc_mat[2],acc_mat[3],acc_mat[0]=func.accuracy(cm_test,y_test,y_predt)
                acc_mat[0]=(totalacc_val+totalacc_test)/2
                acc_mat[1]=(lessacc_val+lessacc_test)/2
                acc_mat[2]=(betweenacc_val+betweenacc_test)/2
                acc_mat[3]=(greatacc_val+greatacc_test)/2
                if acc_mat[0]>topminor_acc[0]:
                    topminor_acc[:]=acc_mat[:]
                    topminor_index[:]=np.empty([net_max_size],'O')
                    topminor_index[:]=index_mat[:]
                i+=1
                index_mat=np.empty([net_max_size],'O')
            if topminor_acc[0]>top3_acc[2,0]:
```

```
            top3_acc[2,:]=topminor_acc[:]
            top3_index[2,:]=np.empty([net_max_size+5],'O')
            top3_index[2,0:topminor_index.shape[0]]=topminor_index[:]
            for j in range(1,-1,-1):
                if topminor_acc[0]>top3_acc[j,0]:
                    top3_acc[j+1,:]=top3_acc[j,:]
                    top3_index[j+1,:]=np.empty([net_max_size+5],'O')
                    top3_index[j+1,:]=top3_index[j,:]
                    top3_acc[j,:]=topminor_acc[:]
                    top3_index[j,:]=np.empty([net_max_size+5],'O')
                    top3_index[j,0:topminor_index.shape[0]]=topminor_index[:]
                else:
                    break
        delete=np.where(X_train_temp_labels != topminor_index)
        X_train=np.delete(X_train,net_ins[delete[0]],1)
        X_val=np.delete(X_val,net_ins[delete[0]],1)
        X_test=np.delete(X_test,net_ins[delete[0]],1)
    else:
        net_ins=ins[0,j*net_max_size:ins.shape[1]]
        X_train_temp_labels=X_train_copy[0,net_ins]
        X_train_temp_data=X_train_copy[1:X_train_copy.shape[0],net_ins].astype(np.int)
        X_val_temp_data=X_val_copy[1:X_val_copy.shape[0],net_ins].astype(np.int)
        X_test_temp_data=X_test_copy[1:X_test_copy.shape[0],net_ins].astype(np.int)
        topminor_acc=np.zeros([4])
        topminor_index=np.empty([net_max_size],'O')
        acc_mat=np.zeros([4])
        index_mat=np.empty([net_max_size],'O')
        i=1
        while i<2**X_train_temp_data.shape[1]:
            num_bits=int(math.floor(math.log2(i))+1)
            bits=[(i >> bit) & 1 for bit in range(num_bits - 1, -1, -1)]
```

```
                bits.reverse()
                result=np.where(np.array(bits) == 1)
                index_mat[result[0]]=X_train_temp_labels[result[0]]
                nb=clf.fit(X_train_temp_data[:,result[0]],y_train)
                y_predv=nb.predict(X_val_temp_data[:,result[0]])
                y_predt=nb.predict(X_test_temp_data[:,result[0]])
                cm_val=confusion_matrix(y_val,y_predv)
                cm_test=confusion_matrix(y_test,y_predt)

lessacc_val,betweenacc_val,greatacc_val,totalacc_val=func.accuracy(cm_val,y_val,y_predv)

lessacc_test,betweenacc_test,greatacc_test,totalacc_test=func.accuracy(cm_test,y_test,y_predt)

#acc_mat[1],acc_mat[2],acc_mat[3],acc_mat[0]=func.accuracy(cm_test,y_test,y_predt)
                acc_mat[0]=(totalacc_val+totalacc_test)/2
                acc_mat[1]=(lessacc_val+lessacc_test)/2
                acc_mat[2]=(betweenacc_val+betweenacc_test)/2
                acc_mat[3]=(greatacc_val+greatacc_test)/2
                if acc_mat[0]>topminor_acc[0]:
                   topminor_acc[:]=acc_mat[:]
                   topminor_index[:]=np.empty([net_max_size],'O')
                   topminor_index[:]=index_mat[:]
                i+=1
                index_mat=np.empty([net_max_size],'O')
             if topminor_acc[0]>top3_acc[2,0]:
                top3_acc[2,:]=topminor_acc[:]
                top3_index[2,:]=np.empty([net_max_size+5],'O')
                top3_index[2,0:topminor_index.shape[0]]=topminor_index[:]
                for j in range(1,-1,-1):
                   if topminor_acc[0]>top3_acc[j,0]:
                      top3_acc[j+1,:]=top3_acc[j,:]
```

```
                top3_index[j+1,:]=np.empty([net_max_size+5],'O')
                top3_index[j+1,:]=top3_index[j,:]
                top3_acc[j,:]=topminor_acc[:]
                top3_index[j,:]=np.empty([net_max_size+5],'O')
                top3_index[j,0:topminor_index.shape[0]]=topminor_index[:]
            else:
                break
        delete=np.where(X_train_temp_labels !=
topminor_index[0:X_train_temp_labels.shape[0]])
        X_train=np.delete(X_train,net_ins[delete[0]],1)
        X_val=np.delete(X_val,net_ins[delete[0]],1)
        X_test=np.delete(X_test,net_ins[delete[0]],1)

    X_train_temp_labels=X_train[0,:]
    X_train_temp_data=X_train[1:X_train.shape[0],:].astype(np.int)
    X_val_temp_data=X_val[1:X_val.shape[0],:].astype(np.int)
    X_test_temp_data=X_test[1:X_test.shape[0],:].astype(np.int)
    acc_mat=np.zeros([4])
    index_mat=np.empty([X_train.shape[1]],'O')
    i=1

    while i<2**X_train.shape[1]:
        num_bits=int(math.floor(math.log2(i))+1)
        bits=[(i >> bit) & 1 for bit in range(num_bits - 1, -1, -1)]
        bits.reverse()
        result=np.where(np.array(bits) == 1)
        index_mat[result[0]]=X_train_temp_labels[result[0]]
        nb=clf.fit(X_train_temp_data[:,result[0]],y_train)
        y_predv=nb.predict(X_val_temp_data[:,result[0]])
        y_predt=nb.predict(X_test_temp_data[:,result[0]])
        cm_val=confusion_matrix(y_val,y_predv)
```

```
        cm_test=confusion_matrix(y_test,y_predt)

lessacc_val,betweenacc_val,greatacc_val,totalacc_val=func.accuracy(cm_val,y_val,y_predv)

lessacc_test,betweenacc_test,greatacc_test,totalacc_test=func.accuracy(cm_test,y_test,y_predt)
        #acc_mat[1],acc_mat[2],acc_mat[3],acc_mat[0]=func.accuracy(cm_test,y_test,y_predt)
        acc_mat[0]=(totalacc_val+totalacc_test)/2
        acc_mat[1]=(lessacc_val+lessacc_test)/2
        acc_mat[2]=(betweenacc_val+betweenacc_test)/2
        acc_mat[3]=(greatacc_val+greatacc_test)/2
        if acc_mat[0]>top3_acc[2,0]:
            top3_acc[2,:]=acc_mat[:]
            top3_index[2,:]=np.empty([net_max_size+5],'O')
            top3_index[2,0:index_mat.shape[0]]=index_mat[:]
            for j in range(1,-1,-1):
                if acc_mat[0]>top3_acc[j,0]:
                    top3_acc[j+1,:]=top3_acc[j,:]
                    top3_index[j+1,:]=np.empty([net_max_size+5],'O')
                    top3_index[j+1,:]=top3_index[j,:]
                    top3_acc[j,:]=acc_mat[:]
                    top3_index[j,:]=np.empty([net_max_size+5],'O')
                    top3_index[j,0:index_mat.shape[0]]=index_mat[:]
                else:
                    break
        i+=1
        index_mat=np.empty([X_train.shape[1]],'O')
    for u in range(top3_acc.shape[0]):
        if top3_acc[u,0]>top3total_acc[2,0]:
            top3total_acc[2,:]=top3_acc[u,:]
            top3total_index[2,:]=np.empty([net_max_size+5],'O')
            top3total_index[2,0:top3_index[u,:].shape[0]]=top3_index[u,:]
```

```
        for j in range(1,-1,-1):
            if top3_acc[u,0]>top3total_acc[j,0]:
                top3total_acc[j+1,:]=top3total_acc[j,:]
                top3total_index[j+1,:]=np.empty([net_max_size+5],'O')
                top3total_index[j+1,:]=top3total_index[j,:]
                top3total_acc[j,:]=top3_acc[u,:]
                top3total_index[j,:]=np.empty([net_max_size+5],'O')
                top3total_index[j,0:top3_index[u,:].shape[0]]=top3_index[u,:]
            else:
                break
    else:
        break

pd.DataFrame(top3_acc).to_csv(path_or_buf='D:\\R Thesis
Files\\top3_acc.csv',header=False,index=False)
pd.DataFrame(top3_index).to_csv(path_or_buf='D:\\R Thesis
Files\\top3_index.csv',header=False,index=False)
```

REFERENCES

[1] Bathla, R. (2019). Personal interview.

[2] Laha, D., Ren, Y., & Suganthan, P. N. (2015). Modeling of steelmaking process with effective machine learning techniques. *Expert systems with applications*, *42*(10), 4687-4696.

[3] Li, F., Wu, J., Dong, F., Lin, J., Sun, G., Chen, H., & Shen, J. (2018). Ensemble Machine Learning Systems for the Estimation of Steel Quality Control. In *2018 IEEE International Conference on Big Data (Big Data)* (pp. 2245-2252). IEEE.

[4] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32.

[5] Genuer, R., Poggi, J. -M., & Tuleau-Malot, C. (2008). Random forests: Some methodological insights. arXiv: 0811.3619.

[6] Jalali-Heravi, M., Asadollahi-Baboli, M., & Shahbazikhah, P. (2008). QSAR study of Heparanase inhibitors activity using artificial neural networks and Levenberg– Marquardt algorithm. *European Journal of Medicinal Chemistry*, 43, 548–556.

[7] Marquardt, D. W. (1963). An algorithm for least-squares estimation of non-linear parameters. *Journal Society Industrial Applied Mathematics*, 11, 431–441.

[8] Haykin, S. (2009). *Neural networks and learning machines*. Pearson Education Inc.

[9] Hagan, M. T., & Menhaj, M. B. (1994). Training feed forward networks with the Marquardt algorithm. *IEEE Transactions on Neural Networks*, 5, 989–993.

[10] Kasabov, N., & Song, Q. (2002). DENFIS: Dynamic evolving neural-fuzzy inference system and its application for time-series prediction. *IEEE Transactions on Fuzzy Systems*, 10, 144–154.

[11] Takagi, T., & Sugeno, M. (1985). Fuzzy identification of systems and its applications to modeling and control. *IEEE Transactions on Systems, Man, Cybernetics*, 15, 116–132.

[12] Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273–297.

[13] Basak, D., Pal, S., & Patranabis, D. C. (2007). Support vector regression. *Neural Information Processing Letters and Reviews*, 11(10), 203–224.

[14] Zhou, Z. H. (2009). Ensemble Learning. *Encyclopedia of biometrics*, *1*, 411-416.

[15] Zhou, Z. H. (2012). *Ensemble methods: foundations and algorithms*. CRC press.

[16] Zhou, Z. H., Wu, J., & Tang, W. (2002). Ensembling neural networks: many could be better than all. *Artificial intelligence*, *137*(1-2), 239-263.

[17] Krogh, A., & Vedelsby, J. (1995). Neural network ensembles, cross validation, and active learning. In *Advances in neural information processing systems* (pp. 231-238).

[18] Draper, N. R.; Smith, H. (1998). *Applied Regression Analysis*. Wiley-Interscience.

[19] Hyndman, Rob J.; Koehler, Anne B. (2006). "Another look at measures of forecast accuracy". *International Journal of Forecasting*. 22 (4): 679–688.

[20] Gurney, Kevin (1997). *An introduction to neural networks*. UCL Press.

[21] Stocco, Andrea; Lebiere, Christian; Anderson, John R. (2010). "Conditional Routing of Information to the Cortex: A Model of the Basal Ganglia's Role in Cognitive Coordination". *Psychological Review*.

[22] McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". *Bulletin of Mathematical Biophysics*.

[23] Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, in PMLR* 9:249-256

[24] Bre, Facundo & Gimenez, Juan & Fachinotti, Víctor. (2017). Prediction of wind pressure coefficients on building surfaces using Artificial Neural Networks. Energy and Buildings. 158. 10.1016/j.enbuild.2017.11.045.

[25] Zell, Andreas (2003). "Chapter 5.2". *Simulation of Neural Networks* (1st ed.). Addison-Wesley.

[26] Dawson, Christian W (1998). "An artificial neural network approach to rainfall-runoff modelling". *Hydrological Sciences Journal*. 43 (1): 47–66.

[27] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *2015 IEEE International Conference on Computer Vision (ICCV)*. doi:10.1109/iccv.2015.123

[28] Haykin, S. (1999). *Neural networks: a comprehensive foundation*. Prentice Hall.

[29] Ioannou, Yani. (2017). Structural Priors in Deep Neural Networks. 10.17863/CAM.26357.

[30] Cybenko, G. (1989) “Approximations by superpositions of sigmoidal functions”, *Mathematics of Control, Signals, and Systems*, 2 (4), 303-314

[31] Nair, V., & Hinton, G.E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *ICML*.

[32] Hochreiter, S., Bengio, Y., Frasconi, P. & Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C. Kremer & J. F. Kolen (ed.), *A Field Guide to Dynamical Recurrent Neural Networks*, IEEE Press.

[33] Lehmann, E. L.; Casella, George (1998). *Theory of Point Estimation* (2nd ed.). New York: Springer.

[34] Lemaréchal, C. (2012). "Cauchy and the Gradient Method". *Doc Math Extra*: 251–254.

[35] Bhattarai, S. (2018). What is Gradient Descent in machine learning?

[36] Bachman, David (2007), *Advanced Calculus Demystified*, New York: McGraw-Hill

[37] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron (2016). "6.5 Back-Propagation and Other Differentiation Algorithms". *Deep Learning*. MIT Press. pp. 200–220.

[38] Nielsen, Michael A. (2015). "Chapter 2: How the backpropagation algorithm works". *Neural Networks and Deep Learning*. Determination Press.

[39] Li, Y.; Fu, Y.; Li, H.; Zhang, S. W. (2009). *The Improved Training Algorithm of Back Propagation Neural Network with Self-adaptive Learning Rate*. *2009 International Conference on Computational Intelligence and Natural Computing*. 1. pp. 73–76.

[40] Kingma, D. P., & Ba, J. L. (2015). "Adam: A Method for Stochastic Optimization." *International Conference on Learning Representations*, 1–13.

[41] Hinton, Geoffrey; Sejnowski, Terrence (1999). *Unsupervised Learning: Foundations of Neural Computation*. MIT Press.

[42] Vaseekaran, G. (2018). Machine Learning: Supervised Learning vs Unsupervised Learning.

[43] Ojha, Varun Kumar; Abraham, Ajith; Snášel, Václav (2017). "Metaheuristic design of feedforward neural networks: A review of two decades of research". *Engineering Applications of Artificial Intelligence*. 60: 97–116.

[44] Ioffe, Sergey; Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift".

[45] Refsgaard, A. (2019). Looking inside neural nets.

[46] Han, Jiawei; Kamber, Micheline; Pei, Jian (2011). "Data Transformation and Data Discretization". *Data Mining: Concepts and Techniques*. Elsevier.

[47] Derrick, T.R. and Thomas, J.M. (2004). “Time-Series Analysis: The Cross-Correlation function”. *Innovative Analyses of Human Movement*, Human Kinetics Publishers, Champaign, Illinois, 189-205.

[48] Hinton, G. E.; Osindero, S.; Teh, Y. W. (2006). “A Fast Learning Algorithm for Deep Belief Nets”. *Neural Computation*. 18 (7): 1527–1554.

[49] Nie, Yali. (2018). A Multi-stage Convolution Machine with Scaling and Dilation for Human Pose Estimation 사람 자세 추정을 위한 스케일링 및 확장 기반 다단 콘볼루션 머신. 10.13140/RG.2.2.34552.96002.

[50] Ripley, B.D. (1996) *Pattern Recognition and Neural Networks*, Cambridge: Cambridge University Press.

[51] Bishop, C.M. (1995), *Neural Networks for Pattern Recognition*, Oxford: Oxford University Press.

[52] Dumitru, C., & Maria, V. (2013). Advantages and Disadvantages of Using Neural Networks for Predictions. *Ovidius University Annals, Series Economic Sciences, 13*(1).

[53] Pearl, Judea (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press.

[54] Piryonesi S. Madeh; El-Diraby Tamer E. (2020). "Role of Data Analytics in Infrastructure Asset Management: Overcoming Data Size and Quality Problems". *Journal of Transportation Engineering, Part B: Pavements*. **146** (2): 04020022.

[55] Rish, Irina (2001). *An empirical study of the naive Bayes classifier*. IJCAI Workshop on Empirical Methods in AI.

[56] Ali, Waleed & Shamsuddin, Siti Mariyam & Ismail, Abdul Samad. (2012). Intelligent Naïve Bayes-based approaches for Web proxy caching. Knowledge-Based Systems. 31. 162–175. 10.1016/j.knosys.2012.02.015.

[57] Cox, D.R. & Wermuth, N. (1996). *Multivariate Dependencies - Models, Analysis and Interpretation*. Chapman & Hall, London.

[58] Dekking, F.M. (Frederik Michel), 1946- (2005). *A modern introduction to probability and statistics: understanding why and how*. Springer.

[59] Grinstead, Charles M.; Snell, J. Laurie (2009). "Conditional Probability - Discrete Conditional". *Grinstead & Snell's Introduction to Probability*. Orange Grove Texts.

[60] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, *12*, 2825-2830.

[61] Lemaître, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, *18*(1), 559-563.

[62] Wagstaff, K., Cardie, C., Rogers, S., & Schrödl, S. (2001). Constrained k-means clustering with background knowledge. In *Icml* (Vol. 1, pp. 577-584).

[63] Marco Scutari, Jean-Baptiste Denis. (2014) Bayesian Networks with Examples in R. Chapman and Hall, Boca Raton.

[64] Marco Scutari (2010). "Learning Bayesian Networks with the bnlearn R Package". *Journal of Statistical Software*, 35(3), 1-22.

[65] Chattopadhyay, A., Manupriya, P., Sarkar, A., & Balasubramanian, V. N. (2019). Neural network attributions: A causal perspective. *arXiv preprint arXiv:1902.02302*.