

THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

Department of Mechanical Engineering

Brain Impact Analysis from Overpressure Sources Through Machine Learning Based on
Explosion Simulations and Wearable Blast Gauges

JACKSON MACKAY
SPRING 2022

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Mechanical Engineering
with honors in Mechanical Engineering

Reviewed and approved* by the following:

Reuben H. Kraft
Associate Professor of Mechanical Engineering
Thesis Supervisor

Daniel Cortes
Assistant Professor of Mechanical Engineering
Honors Advisor

* Electronic approvals are on file.

ABSTRACT

Wearable sensors gauges are increasing in demand to be worn by soldiers in combat in order to track the overpressures that they experience. These sensors create the potential to analyze blast overpressures and their effects on the brain. One company, BlackBox Biometrics, have popularized these wearable gauges with sets of three typically being worn on the chest, one shoulder and the back of soldier's helmets. The problem with this is that the overpressure data is tracked at these locations, and not the face which is needed for better brain computational analysis. Due to this, there is great interest in research on how to transform these chest-shoulder-helmet datums into accurate facial overpressures. Prior research on this topic suggests that machine learning algorithms can accurately predict these sought after overpressures. In this paper we utilize a blast simulation application called Viper::Blast to simulate combat environments in order to collect a library of overpressure data to train linear regression, ridge regression, and deep learning algorithms for accurate prediction of facial overpressure to be used with BlackBox Biometric gauge data.

TABLE OF CONTENTS

LIST OF FIGURES	iii
LIST OF TABLES	iv
ACKNOWLEDGEMENTS	v
Chapter 1 Introduction	1
1.1 Problem Statement	1
1.2 Motivation	1
Chapter 2 Literature Review	2
2.1 What is TBI and ICP?	2
2.1.1 Causation of Traumatic Brain Injury	3
2.1.2 Science of Traumatic Brain Injury	4
2.1.3 Intracranial Wave Mechanics Due to Blasts	5
2.2 Diagnostics of Traumatic Brain Injury	7
2.3 History of Intracranial Pressure Monitoring Technology	8
2.4 History of Blast Gauges	10
2.4.1 How do Blast Gauges Work?	10
2.4.2 Blast Gauge Positioning Data Gap	11
2.5 Past Thesis Work	11
2.6 What is Machine Learning (ML) and Computational Modeling?	12
2.7 AWS Computing and Viper	14
Chapter 3 Materials and Methodology	17
3.1 Viper CFD Simulation Software	17
3.1.1 Viper Inputs and Considerations	17
3.1.2 Monte Carlo	20
3.2 Models	22
3.2.1 Open-Field Geometry	23
3.2.2 Sniper Geometry	24
3.2.3 Breacher Geometry	25
3.2.4 Crouched Geometry	26
3.3 Computational Run Scenarios	27
3.3.1 Amazon S3 Bucket Transfer Code	27
3.3.2 Obstacles	29
3.3.3 Blast Variation of Location and Magnitude for Scenarios	31
3.4 Machine Learning Code	32
Chapter 4 Results	37
Chapter 5 Discussion	42

Chapter 6 Conclusion.....	44
Chapter 7 Future Work	45
Appendix A Python Code for Machine Learning Algorithms.....	46

LIST OF FIGURES

Figure 1: Pressure versus time curve of a typical Friedlander blast wave model.	6
Figure 2. Different ICP Monitoring device insertion locations. Source is open-access.	8
Figure 3: The Blast Gauge System by BlackBox Biometrics [15].	11
Figure 4: Basic breakdown of a decision tree showcasing root, interior, and leaf nodes used to make decisions.	13
Figure 5. Interface of Viper::Blast, a weapons effect simulator utilized for close proximity blast explosions. The software imports CAD models of soldiers and uses sensors (yellow dots) placed all around their heads to track pressure data from blast explosion sites (red dot).	15
Figure 6. User input data example in the Viper interface for the Domain and Charge for CFD blast simulations.....	18
Figure 7. Time history output locations example table for the Sniper geometry model with all thirteen sensors labeled	19
Figure 8. All thirteen overpressure time history output sensor locations on an open-field model.	19
Figure 9. Monte Carlo example user input for the Breacher geometry.....	21
Figure 10. Example file collection code for the Sniper geometry model.	22
Figure 11. Open-field geometry model and box obstacle model shown in Blender.....	23
Figure 12. Sniper geometry model shown in Blender.....	24
Figure 13. Breacher geometry model shown in Blender.....	25
Figure 14. Crouched geometry model and wall obstacle shown in blender.	26
Figure 15: Cygwin64 Terminal commands for running the File Collection Code.	27
Figure 16: Cygwin64 Terminal commands for exporting files to AWS S3 Bucket	28
Figure 17: Isotropic view of Open-Field soldier geometry for clear view of the box obstacle and its four distinct locations in blender.	29
Figure 18: Breacher geometry model with the additional riot shield obstacle.....	30
Figure 19. Viper blast simulation scenario for the breacher geometry with a double wall and shield obstacle showcasing the pressure waves from the blast.	31
Figure 20. Lines of code to define the dataset from the s3 Bucket and read in the overpressure files.....	33

Figure 21. Code to define the number of people in the simulation model.....	34
Figure 22. For loop Python code that extracts the peak overpressure values from Viper simulation files.....	34
Figure 23. For loop to define names for sensor overpressure data from Viper simulation database.....	35
Figure 24. Creation of three data sets containing a series of peak overpressure information by execution of the create_df function.....	35
Figure 25. Python code to define target and predictor variables for the ML algorithms.	36
Figure 26. Python code to split the overpressure data library into training and validation sets randomly.	36
Figure 27. Viper Blast example Overpressure vs Time graph showcasing all thirteen sensors' pressures in real time.....	37
Figure 28. Series of peak overpressure prediction vs scenario count plots testing the feasibility of the linear and ridge regression ML models.	38
Figure 29. Output peak pressure prediction vs scenario count graph for all 10 non-targeted sensors for linear, ridge, and Keras regression ML algorithms.....	40

LIST OF TABLES

Table 1: Monte-Carlo generated distribution example for five scenarios for the breacher model with variation of explosive mass and location.	21
Table 2. R^2 values for linear, ridge, and deep learning Keras regression models for all 10 non-targeted sensor locations.	42

ACKNOWLEDGEMENTS

I would like to start off by thanking Dr. Reuben Kraft for providing me with the opportunity to work side by side with him and conduct this research. I would like to thank him for his extremely valuable help with the coding aspects of this thesis. I would also like to thank Vikraman Subramani for his guidance and prior coding work for this research. Additionally, I would like to acknowledge both Mykhailo Havrylets and Ahmed Aly for their CAD modeling work. I would also like to thank Dr. Daniel Cortes for his assistance and overview of this honors thesis.

Chapter 1

Introduction

1.1 Problem Statement

Using machine learning algorithms based upon Viper simulation data to predict and analyze facial overpressures from body worn sensors in military blast scenarios.

1.2 Motivation

Today, it is still highly unknown how to treat and prevent brain trauma effectively occurring from blast explosions in close proximity, i.e., in warfare scenarios. In an effort to understand brain trauma, brain neural analysis methods must be utilized. Currently, there is no way of accurately measuring facial overpressures in blast scenarios which is necessary for proper brain analysis. Due to the severity and extreme incidence of head trauma, especially in war, it is exceedingly becoming of importance that an accurate way of measuring facial overpressures be established.

Chapter 2

Literature Review

2.1 What is TBI and ICP?

Traumatic brain injury (TBI) is an unfortunate condition that results in loss of motor, cognitive, or memory functions in the brain due to head impacts or blast overpressures on the head/brain [1]. TBI is a huge problem around the world today as it is one of the leading causes of mortality among children and adolescents [2]. Not only this, but it is extremely prevalent in athletes, soldiers, and even everyday drivers. Currently, there are over 5 million people who live with TBI in the United States alone [1]. According to the CDC, 53,000 people die from TBI-related injuries every year which contributes to 30.5% of all injury-related deaths [3]. With no signs of halting, TBI related cases continue to rise and become more and more of a global problem.

Although the incidence of TBI is increasing worldwide, the understanding and research into long-term outcomes is still limited. Prevention and treatment are becoming more and more of a necessity for TBI as cases increase. In order to research and test ways for both, it has been hypothesized that monitoring of intracranial pressure (ICP) can facilitate and help treat or prevent traumatic brain injury [4].

ICP is the pressure that the brain experiences from things like blows to the head or blast explosions. Monitoring of ICP allows researchers to analyze how the brain reacts to pressure and what damage it can cause [2]. This data can directly translate to TBI and how it occurs from overpressures. In the past, pediatric TBI centers across Europe and the US have conducted TBI monitoring studies which have unanimously reported the use of an ICP threshold of 20 mmHg,

meaning ICP values above this can cause brain damage [2]. In order to better understand traumatic brain injury and how intracranial pressure monitoring is related, we need to understand the causes of TBI.

2.1.1 Causation of Traumatic Brain Injury

Traumatic brain injury can be caused by any force or impact on the head/brain, so naturally there are many different ways this can occur. Falls among children (0 to 17 years) and older adults (65 years and older) accounted for 48% of all TBI-related ED visits in the same year in the US [3]. Another 17% of ED visits was caused by being struck with or against an object [3]. For just general TBI cases, 21% are caused by sports or recreational activities among American children. [5]. And more specifically, 50% of those cases occur during bicycling, skateboarding, or skating [5]. These statistics are alarming in their own right, but are reported in a country with no real warzones.

An overlooked statistic of TBI related cases and something that isn't reported in US hospitals normally is from explosive mechanisms and blast overpressures in military environments. In the wars in Iraq and Afghanistan, explosive mechanisms or improvised explosive devices (IEDs) accounted for 70-75% of allied military killed or wounded – the leading cause of casualty [6]. For those that survived blasts or nearby explosions, 35% of allied soldier deaths were caused by TBI in 2009 [6]. The US Department of Defense estimated the total number of medically diagnosed cases of TBI in the US military from 2000-2010 was 202,281 [6]. Due to the extreme numbers of TBI related cases and deaths from the Iraq and Iran

wars, there has been a huge push for research into TBI and the science behind it in order to better understand it.

2.1.2 Science of Traumatic Brain Injury

The mechanism of production of TBI differs amongst the causation and is dependent upon the physics of the brain [7]. Modeling of human brain physics first dates back to the 1940s where Halbourn, an Oxford research physicist, stated that the behavior of the skull and brain during and immediately after impact was determined by five physical properties of the skull/brain and Newton's laws of motion [8]. These physical properties are as follows: the brain has comparatively uniform density, it's extremely incompressible, it has a very small modulus of rigidity, the skull has a comparably large modulus of rigidity, and the shape and size of the brain/skull [8]. On the basis of these five properties and modeling experiments, Halbourn concluded that the amount of pulling apart of the brain particles during impact was proportionally related to the shear-strain occurring within the brain [8]. This directly leads to the discovery that shear-strains are the cause of injury to the brain. Halbourn found that brain injury naturally falls into two categories due to mathematical theory: localized injury due to skull distortion, and injury due to rotation [8]. As one can imagine, if the skull is distorted or fractured, major shear-strains can develop in the brain around the fracture causing massive damage. In this case, hemorrhaging is common and can cause internal bleeding and quickly become life threatening [8].

In the case of non-distortion of the skull, brain injuries are rooted in the rotational acceleration of the brain/skull. The reason that rotational acceleration is the cause of injury is

because the shear-strains produced by linear acceleration are relatively small and can be neglected [8]. This can easily be seen by filling a flask with water where moving it linearly produces little effect on the water whilst rotating the flask tends to leave the water behind as water particles attached to the surface of the flask separate from neighboring particles not attached to the flask [8]. This is obviously an exaggerated model but holds true to the effects on the brain. Halbourn went on to actually run experiments on the shear-strain concentrations in the brain due to different rotations caused by different blows to the head. The most striking aspect of this study is the large shear-strain concentrations in the anterior temporal lobe compared to the absence in the cerebellum. This can be explained by the fact that the skull is somewhat attached to the brain at this location and causes massive shearing when rotating at a high enough velocity [8]. Additionally, a force impulse on the head creates non-linear shear waves which propagate at different speeds in different regions therefore creating localized strain concentrations in different regions [7].

The development of these shear-strains can be applied to many different types of brain injuries and their causation. Although the physics discussed above was for direct impacts on the head, it can still be applied for indirect pressures on the head, such as an explosion/blast. Blast pressure waves act slightly different then the shock waves produced from direct blows and therefore must be modeled differently within the brain.

2.1.3 Intracranial Wave Mechanics Due to Blasts

For blast pressure waves, the mechanism is less understood, but can be related to the Friedlander waveform which causes rapid rise in pressure to a peak over-pressure, followed by a

negative pressure drop period of under-pressure. After passing through the skull, this negative pressure drop can lead to cavitation within the head/brain tissues [6]. This over to under pressure can be seen in Figure 1 along with the Friedlander equation.

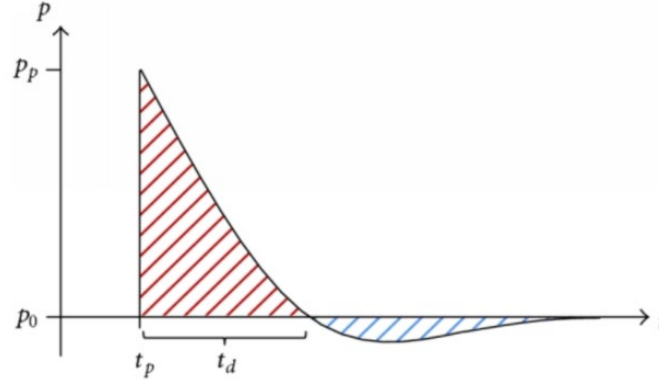


Figure 1: Pressure versus time curve of a typical Friedlander blast wave model.

$$P_s(t) = P_{so} \left(1 - \frac{t}{t_0}\right) e^{-b \frac{t}{t_0}} \quad (1)$$

In the Friedlander equation, there is a decay coefficient b defined which is very important to how these waves function in different materials [10]. Blast waves vary depending upon the material they are in or if they are incident or reflected, and the coefficient b allows this equation to be dynamic for different cases [10]. For the case of blast waves, brain shear-strains develop not only from rotational acceleration but also from the interaction of these blast waves with brain composition.

Paul Taylor, a researcher at the Sandia National Laboratories, created a model of these blast waves within the brain in order to test this interaction [11]. The brain model was much more complex and realistic than Halbourn's model in the 1940s. It included cerebrospinal fluid and air in the sinuses in addition to the skull and brain for more accurate results [11]. The model was run through three different orientations of blast exposure simulations with a time interval of only 2 ms as the majority of the intracranial wave mechanics was observed to occur almost

immediately [11]. For the tests, blasts were placed in the front, back, and right side of the model. The pressure within the brain closest to the blasts was recorded the highest and the opposite end of the brain correlated to the highest volumetric tension [11].

This study confirmed the fact that blast waves have an extreme effect on the brain before any brain acceleration even occurs. Due to this, the initial blast wave interaction within the brain must be included with the shear-strains developed from rotational acceleration. This helps researchers and doctors diagnose TBI with more specificity to the amount of shear-strain and what is causing it.

2.2 Diagnostics of Traumatic Brain Injury

A commonly used diagnostic scale for TBI is the Glasgow Coma scale. The scale is weighed upon three aspects of responsiveness: motor, verbal, and eye-opening responses [12]. The levels of each respective parameter are scored from a minimum value of 1 (no response) up to a maximum value of 4 for eye-opening response, 5 for verbal response, and 6 for motor response [12]. Thus, the Glasgow Coma scale has values between 3 (worst) and 15 (best) [12]. Each numbered score for each parameter has a descriptive category that allows for ease of scoring. In addition to this scale, TBI can be categorized by the cause of injury. The three main categories of TBI are closed head (CHTBI), penetrating (PTBI), and explosive blast (EBTBI) [6]. Within these groupings, a range of severity is noted ranging from mild (GCS13-15) to moderate (GCS9-12) to severe (GSC3-8) with mild meaning just a brief change of consciousness (concussion), severe meaning a significant period of unconsciousness/memory loss, and moderate lying everywhere in between the two [3], [12]. Although the Glasgow Coma scale can

get specific, categorizing and diagnosing of TBI can be further broken down by the use of ICP monitoring.

2.3 History of Intracranial Pressure Monitoring Technology

Intracranial pressure (ICP) monitoring has been theorized and tested for decades. In 1951, the first data set published on ICP measurement was by two French doctors named Guillaume and Janny [13]. They were able to create continuous ICP monitoring through an electronic magnetic transducer (EMT) [13]. This allowed them to measure the changes in ventricular fluid pressure within the brain. Their findings directly led to the first comprehensive analysis of ICP in patients using curve morphology in 1965 [2]. In both cases, a transducer coupled to an external ventricular drain (EVD) was used to measure ICP with great accuracy [2]. This is an invasive monitoring device that is inserted into the brain at location A shown in Figure 2. This device works to measure the pressure within the brain by connecting a drainage tube from the inside of the brain to the outside of the head to measure the difference between atmospheric pressure and intracranial pressure. It simultaneously drains cerebrospinal fluid (CSF) in order to manage intracranial hypertension, which is basically just an increased overall pressure in the skull [2].

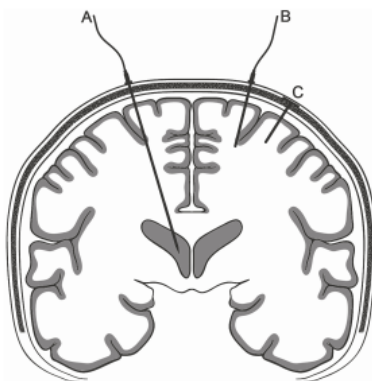


Figure 2. Different ICP Monitoring device insertion locations. Source is open-access.

Due to the extremely invasive nature of the EVD, an overall complication rate of 20-25% has been estimated for it which includes infection, hemorrhage, misplacement, and malfunction [2]. With this in mind, other methods of ICP monitoring have been developed. Parenchymal monitoring is another means of measuring ICP in a less invasive way. It can take on many different forms including fiber optic sensors (Camino ICP Monitor) which transmits light via a fiber optic cable towards a displacement mirror, strain gauge devices (Codman MicroSensor ICP sensor) which measure the resistance change in an inserted transducer correlating to ICP, and pneumatic sensors which make use of a small balloon connected to a catheter placed within the brain to register changes in pressure [2], [14]. Each of these monitoring techniques are used in different parts of the brain and at different depths. The most commonly used are the strain gauge devices which are still inserted into the brain like the EVD, but is instead placed within the frontal lobe at a depth of only 2cm at location B shown in Figure 2 [2].

Another monitoring method, telemetric ICP monitoring, is an even less invasive device that uses a parenchymal strain gauge sensor coupled to a wireless transcutaneous data transmitter in order to wirelessly collect pressure data [2]. The telemetric device is inserted at location C shown in Figure 2. Even though the parenchymal and telemetric devices are less invasive and still very accurate, they also have high complication rates [2]. Due to the invasiveness and high risk of these devices, researchers and doctors are currently looking for more accurate and preferably non-invasive ways to monitor ICP. The recent development of blast gauges is a promising way to monitor overpressures without being inserted in the brain and could be used for future ICP monitoring.

2.4 History of Blast Gauges

Wearable blast gauges for soldiers have had a semi short history. Towards the end of US involvement in the Iraq War, 200,000 US soldiers came home with TBI related injuries without proper data in-battle to know the causes [15]. In light of this, the Defense Advanced Research Projects Agency (DARPA) enlisted the help of the Rochester Institute of Technology (RIT) to develop the Blast Gauge [16]. In 2011, the team developed the Blast Gauge and formed a business called BlackBox Biometrics to manufacture and sell them. This device allowed for measurement of blast overpressures to be analyzed for blast propagation and traumatic brain injury. Today, BlackBox Biometrics have sold 550,000 Blast Gauges [15].

2.4.1 How do Blast Gauges Work?

The Blast Gauge developed by BlackBox Biometrics works in a multi-sensor set meaning one is placed on the back of the helmet, the chest, and a shoulder [15]. It has the ability to record pressure and acceleration data in real-time. There are seven parts worth noting on the Blast Gauge as seen in Figure 3. The main component of the gauge is the sensor dome which has an accelerometer for acceleration data and takes in blast pressure waves to store pressure data. It has a recessed activation button to make sure it doesn't activate/deactivate unwanted and indicator lights of red, green, and yellow to provide instant exposure data [15]. Although it has wireless technology to transfer data, it has a micro-USB port for full time-based data analysis [15]. The Blast Gauge also has an impact resistant casing, a heavy-duty attachment cord for seamless body connections, and a color-coded body placement mark for each respective body part. This device works by tracking triage data from the three body locations through proprietary software which

is very helpful in tracking pressure data near the head/face, but does not track the direct pressures on it. [15].



Figure 3: The Blast Gauge System by BlackBox Biometrics [15].

2.4.2 Blast Gauge Positioning Data Gap

The Blast Gauge is the current best blast pressure tracking system for soldiers, but still has a gap from its data to real facial pressures. Due to the fact that the Blast Gauge is set up on a soldier's chest, head, and shoulder, it tracks pressure data at those exact locations. This data set, called triage data, can help estimate facial overpressures but is far from an exact facial pressure measurement. Currently, there is no accurate transfer function from body sensor to face and no real way to get accurate data on the face from Blast Gauges for brain analysis.

2.5 Past Thesis Work

Lauren Katch, a Schreyer alumnus, previously wrote a thesis on this topic to help determine this transfer function. She focused on developing a geometric transfer function that

would triangulate the data from the chest, shoulder, and helmet to identify the magnitude and location of the blast source [17]. This was called inverse-source localization and was the focus of her thesis. She ran simulations through a blast simulation software known as Kodiak CTH and tracked pressure data on the shoulder, head, and helmet of a soldier model. She fit a geometric model to the data to try and predict the locations of the blast explosions, but this fit model had an average error of 71.23% [17]. She concluded that blast explosions at close-range were too complex for a geometric algorithm. Katch then went on to attempt a different route of data transfer, through machine learning. Instead of predicting the location of the blast, this time she used chest, helmet, and shoulder data to predict nose data. Since most of the thesis work was spent on the geometric algorithm for inverse-source localization, the machine learning approach is limited. However, with just a limited number of simulations, she was able to predict nose pressure data with relatively high accuracy and proved that machine learning could be a viable option [17]. My thesis work aims to build off of Katch's, and create an accurate machine learning algorithm through an immense variety of blast simulations instead of using already developed algorithms like she did.

2.6 What is Machine Learning (ML) and Computational Modeling?

Machine learning is essentially computer programming directly rooted in artificial intelligence (AI). It allows computers to 'learn' and change things without being directly coded to do so [18]. Utilizing machine learning algorithms enables computers and AI to predict things, which can be an extremely useful tool. How machine learning works in theory is simple but in

application requires large and complicated algorithms. Basically, computers are able to learn, predict, and improve by experience through large amounts of tasks, simulations, and data runs. It is quite similar to statistical algorithms by fitting to data [18]. Machine learning is generally classified into two groups: “supervised” and “unsupervised” learning. Supervised learning occurs when the value of the dependent variable is known for each observation [18]. Unsupervised learning occurs when the outcome or value of the dependent variable is not known, and the algorithm attempts to fit data for it through identified natural relationships [18]. One of the most popular machine learning algorithms is decision trees. Decision trees create a series of rules based upon continuous input variables to predict outputs variables through decisions [18].

Figure 4 shows an example of a breakdown of a basic decision tree machine learning algorithm.

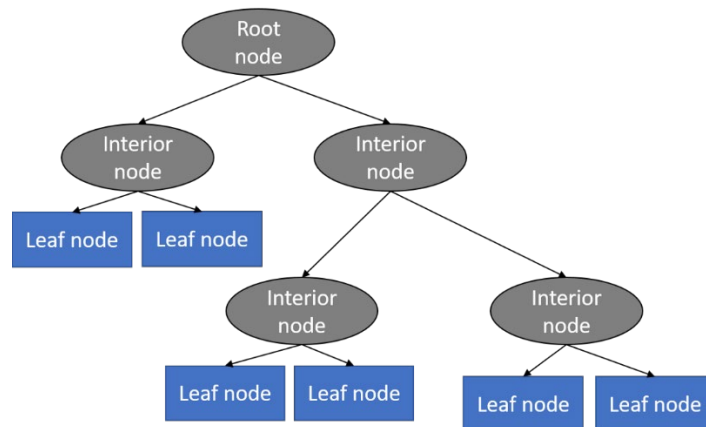


Figure 4: Basic breakdown of a decision tree showcasing root, interior, and leaf nodes used to make decisions.

Decision tree algorithms are generally easy to understand and used frequently. The type of machine learning algorithm that Katch used in her thesis was the random forest which is a type of decision tree algorithm. In any case, the benefit of using machine learning is that it can analyze “Big Data” or extremely large data sets, tasks, simulations, or computational models which computers can run through extremely fast [18].

Computational modeling is the simulation and study of complex systems using computers [19]. This is another extremely useful tool which allows scientists and researchers to simulate thousands of experiments extremely fast through a multi-variable model. Through these computer-generated results, laboratory or field experiments can be identified to be most effective in solving the problem being studied through modeling [19]. This is how Taylor, and the Sandia National Laboratories, was able to run blast simulations on a brain model to determine that blast waves have an effect on ICP in addition to brain rotational acceleration. They modeled a realistic brain in a computer and created a code to run blast simulations to collect ICP data. Katch also made use of computational models of soldiers to collect overpressure data on the chest, head, and shoulder of a soldier for analysis. There are many useful computational modeling applications for blast simulations including the Kodiak CTH used by Katch. For my own work, I will be using a computational modeling platform known as Viper::Blast.

2.7 AWS Computing and Viper

Viper::Blast is a weapon effects simulator used to create computer models to run simulations on. In brief, the application takes an STL file of a Computer Aided Design (CAD) model (of a soldier) then uses set inputs such as boundary conditions, initial blast location, sensor locations, and potential obstacle locations to output pressure data at the sensor locations. The interface for this application is shown in Figure 5.

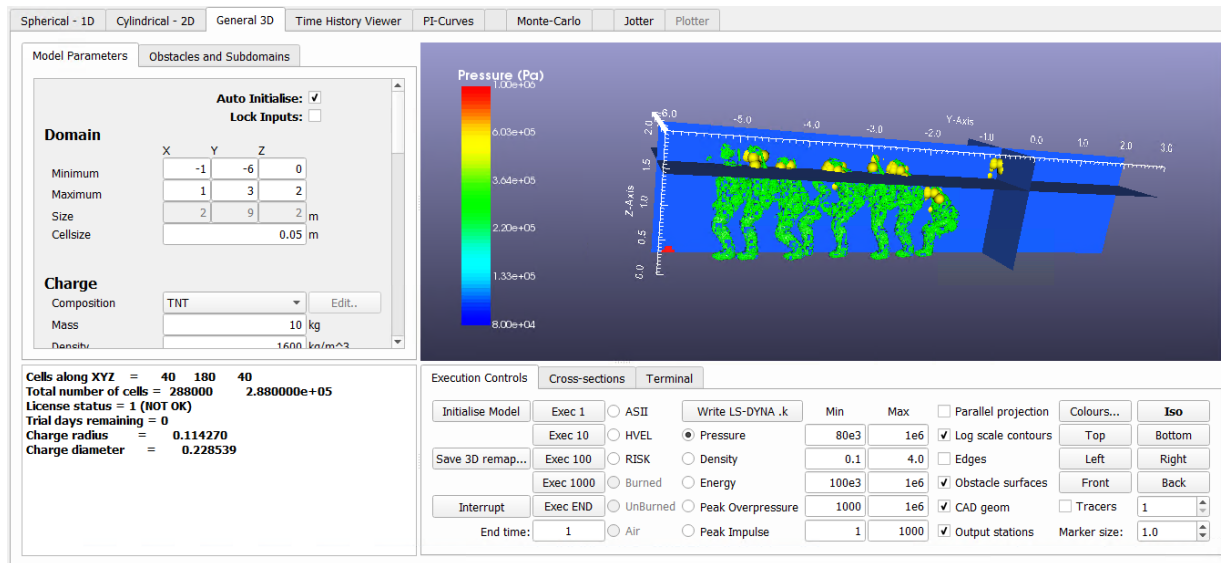


Figure 5. Interface of Viper::Blast, a weapons effect simulator utilized for close proximity blast explosions. The software imports CAD models of soldiers and uses sensors (yellow dots) placed all around their heads to track pressure data from blast explosion sites (red dot).

Every simulation ran on Viper automatically creates an overpressure data output file which compiles all the pressure vs. time data at every sensor location for the simulation run. This application also has a feature called Monte-Carlo (as seen in one of the tabs in Figure 5) which allows for the setup of multiple different simulation runs at once. For example, this feature allows the user to run hundreds of blast simulations in succession with each run having a different input value such as the blast location or an obstacle location. This feature allows for mass simulating to collect tons of different overpressure data files. The Monte-Carlo feature is important because if you were to input these overpressure data files into a machine learning algorithm, the algorithm would be able to predict certain outputs with higher accuracy the more data it collects and uses. The downside of Viper and this feature is that it takes a lot of computing power and usually cannot be done on common laptops and computers.

Amazon Web Services (AWS) is a platform that allows users to gain local access to powerful computers that amazon owns. These supercomputers are more than capable of running

mass simulation programs such as Viper. The way AWS works is through cloud computing, which is basically a wireless connection to IT resources over the internet [20]. So instead of physically owning data centers, servers, computing power, massive storage, and databases, you can simply “connect” to a needed technology service through AWS on a pay-as-you-go basis [20].

In essence, this thesis aims to use computational modeling to simulate blast overpressure scenarios through the application Viper on an AWS computer and then use machine learning to predict and analyze facial overpressures from these simulations. This will allow us to apply our machine learning algorithm to real blast gauge data from BlackBox Biometrics to better estimate human facial overpressures from blasts. With this, researchers and doctors will be able to better track ICP non-invasively to find ways to treat and prevent TBI.

Chapter 3

Materials and Methodology

3.1 Viper CFD Simulation Software

All simulations and collection of overpressure datum in this thesis were ran and extracted from Viper. In order to fully understand the methodology of Viper used in this thesis, a deeper explanation of its functionality must be introduced.

3.1.1 Viper Inputs and Considerations

In order to properly execute CFD blast simulations in Viper, a great deal of inputs and boundary conditions need to be manually set. The most important input is the geometry for which the blast is being set around, which in these cases is .stl files of soldiers in different scenarios. These geometry models will be discussed with more detail in Section 3.2. Besides the model input, there are three other important input categories. For starters, the domain of the blast simulation area must be manually set with considerations of real life applications and what's necessary for proper overpressure data collection. Additionally, the makeup of the explosive charge must be inputted with considerations for composition, mass, density, energy, locations, and shape. For all scenarios, the composition was set to be TNT, the density was kept constant at 1600 kg/m^3 , and the energy produced was set to $4.52 \times 10^6 \text{ J/kg}$. The mass, location, and charge shape were all varied in different ways to produce large sets of different scenarios. An example of these inputs directly in the Viper interface can be seen in Figure 6.

Auto Initialise: ☒
Lock Inputs: ☐

Domain

	X	Y	Z	
Minimum	-1	-6	0	
Maximum	1	3	2	
Size	2	9	2	m
Cellsize				0.05 m

Charge

Composition: TNT Edit..

Mass: 10 kg

Density: 1600 kg/m³

Energy: 4.52e+06 J/kg

Charge Location

X	Y	Z	
0	-5.75	0	m

Shape: Spherical

Radius	Aspect
0.11427	1

Length	Width	Height	
0.1	0.1	0	m

Detonation Location

X	Y	Z
0.0	0.0	0.0

Remap ? ☐ Yes ☒ No Edit..

More charges...

Figure 6. User input data example in the Viper interface for the Domain and Charge for CFD blast simulations

The third important user input is the time history output locations, or just simply the overpressure sensor locations. This is what allows us to set sensors on the body of the soldier geometries. In short, these sensors are created and placed on the geometries directly in Blender, a CAD software that the soldier models were created in. Again, this will be further discussed in Section 3.2. For understanding of Viper user input, these sensors are extracted from Blender in a text file giving label and coordinate information which is then inputted directly into Viper. For all soldier geometries, there are a total of thirteen sensors placed on each soldier. Their locations respective to each person is as follows: left eye, right eye, forehead, nose, left ear, right ear, chin, right cheek, left cheek, chest, back of the helmet, left shoulder, and right shoulder. This is a large

upgrade from the four sensor locations that were used in the simulations that Katch ran in her thesis. The reason for the addition of nine new sensors was to create a more accurate sensor field over the face of the soldier in order to better collect and train data to predict the overpressures across the face, rather than just at the tip of the nose. An example input table of the time history output locations can be seen in Figure 7. These can be visualized on a soldier model in Figure 8 below.

Edit Time History Output Locations

ID	2D	3D	X	Y	Z	Label
1			0.032519	-1.056907	0.489489	eye_left
2			-0.032513	-1.056907	0.489489	eye_right
3			0.000003	-1.067688	0.515007	forehead
4			0.000003	-1.088767	0.453312	nose
5			0.086362	-0.944935	0.481856	ear_left
6			-0.086360	-0.944936	0.481857	ear_right
7			0.000003	-1.062539	0.374010	chin
8			-0.048480	-1.054380	0.447312	cheek_right
9			0.048483	-1.054380	0.447312	cheek_left
10			0.000000	-0.883931	0.134023	chest
11			0.000378	-0.817972	0.520182	helmet_back
12			0.175676	-0.867635	0.394370	shoulder_left
13			-0.181156	-0.855393	0.392426	shoulder_right

Buttons: Add, Import..., Delete

Figure 7. Time history output locations example table for the Sniper geometry model with all thirteen sensors labeled

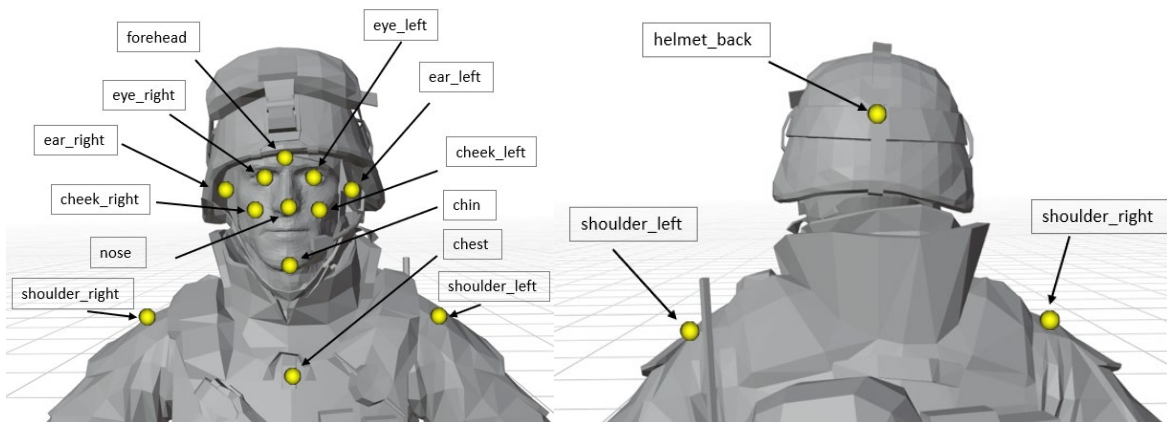


Figure 8. All thirteen overpressure time history output sensor locations on an open-field model.

Besides these inputs, there are other constants and considerations to be noted. All scenarios set the atmospheric pressure and temperature to 101325 Pa and 288 K, respectively. All boundaries of the simulation domain were set to be perfectly transmissible, and all obstacles (including the ground) were set to be perfectly reflective. Creating realistic destruction of obstacles which would allow for some transmissibility is outside the scope of Viper simulations.

3.1.2 Monte Carlo

As mentioned in Section 2.7, the Monte-Carlo feature allows for automated running of variations of inputs in order to create different simulation scenarios. It is the reason why Viper is so good at batch processing and running large amounts of simulation scenarios. In essence, the more data you run through a machine learning algorithm to train it, the more accurate it will become, making the Monte-Carlo feature extremely valuable for this thesis. The focus for different scenarios simulated was varying the charge locations as well as its mass, primarily. This is in addition to other variations such as movable objects, walls, etc. All other variables were kept constant in this feature. In order to vary an input, a mean, standard deviation, and number of scenarios were set for a uniform distribution. An example input interface for the Monte-Carlo feature can be seen in Figure 9.

V Generate scenarios from distributions ×

	MEAN	Standard Deviation	Distribution
Mass	6	2.5	Uniform
TNT Eq	1	0	Uniform
W(user)	1	0	Uniform
Charge X	0	3.5	Uniform
Charge Y	-5	4	Uniform
Charge Z	2.5	1.5	Uniform
HOB_2d	0.3	0	Uniform
dx_1d	0.001	0	Uniform
dx_2d	0.005	0	Uniform
dx_3d	0.08	0	Uniform
Charge_E	4.52e+06	0	Uniform
Charge_RHO	1600	0	Uniform
AB_e	0	0	Uniform
AB_t	0	0	Uniform
Atmos P	101325	0	Uniform
Atmos T	288	0	Uniform

Number of scenarios: 200

OK Cancel

Figure 9. Monte Carlo example user input for the Breacher geometry.

This will create a randomized distributed table of blast scenario runs with the prescribed inputs. The inputs that we wish to vary, explosive mass and location, will be generated in a distribution while all other inputs will remain constant, as shown in Table 1. Two things to note is that the blast location must be within the set domains, and obviously, mass cannot be negative.

	State	Mass	X_3d	Y_3d	Z_3d	HOB_2d	dx_1d	dx_2d	dx_3d	E	Density	AB_e	AB_t	AtmosP	AtmosT
1	0	7.864	-0.76	-11.0	3.95	0.3	0.001	0.005	0.08	4.5e6	1600	1	0	101325	288
2	0	4.823	-1.09	-2.44	1.11	0.3	0.001	0.005	0.08	4.5e6	1600	1	0	101325	288
3	0	3.176	-1.23	0.37	0.72	0.3	0.001	0.005	0.08	4.5e6	1600	1	0	101325	288
4	0	1.107	1.57	-8.34	2.71	0.3	0.001	0.005	0.08	4.5e6	1600	1	0	101325	288
5	0	10.823	-4.72	2.17	4.45	0.3	0.001	0.005	0.08	4.5e6	1600	1	0	101325	288

Table 1: Monte-Carlo generated distribution example for five scenarios for the breacher model with variation of explosive mass and location.

After randomized uniform scenarios are created with this feature, Viper allows you to run all scenarios in succession. Since Viper creates output files automatically, this feature creates folders for all output files as well. The only file we are interested in in these folders is the

overpressure files, so in order to easily extract these files for collection from a large amount of scenario files, a small code is needed. This code, seen in Figure 10, just simply extracts all files from all Viper created scenario folders with the name ‘viper3d_th_overpressure.txt’ (which is the automatic name given to overpressure output files from Viper) and then transfers them to a newly created folder for ease of access. Unfortunately, the downside of this code is that each different geometry model needs its own variation.

```

1  #!/bin/bash
2  rm collected_data_sniper_tnt_mass/*
3  for i in {1..10}
4  do
5      echo "directory: scenario_$i"
6      cp sniper/scenario_$i/viper3d_th_overpressure.txt collected_data_sniper_tnt_mass/viper3d_th_overpressure_sniper_tnt_mass_s$i.txt

```

Figure 10. Example file collection code for the Sniper geometry model.

3.2 Models

In order to generate an exhaustive data library with many different scenarios for better machine learning training, four completely different CAD designs were created. All four were designed by two freelance workers on UpWork that Dr. Kraft hired, Ahmed Aly and Mykhailo Havrylets, as mentioned in the Acknowledgements section in this thesis. They utilized a CAD application called Blender, a comprehensive modeling platform that allows for creating 2D and 3D models. Not only were they able to design and mesh extremely detailed soldier geometries, but they also created sensor tracers in Blender which were used in Viper for simulations. These four CAD models were then used to create 12 distinct testing model variations by addition of obstacles and items. This was done to increase depth of scenarios covered to hopefully create a more accurate and wider scoped ML algorithm. These 12 models are the basis for all 2400 simulation scenarios created for analysis through this thesis.

3.2.1 Open-Field Geometry

The first geometry that was created (and utilized by Katch in her honors thesis) has been labeled as the open-field geometry. This is a basic standing single soldier geometry with a 2m x 2m x 2.5m (height being 2.5m) box obstacle included. The open-field geometry can be seen in Figure 11.

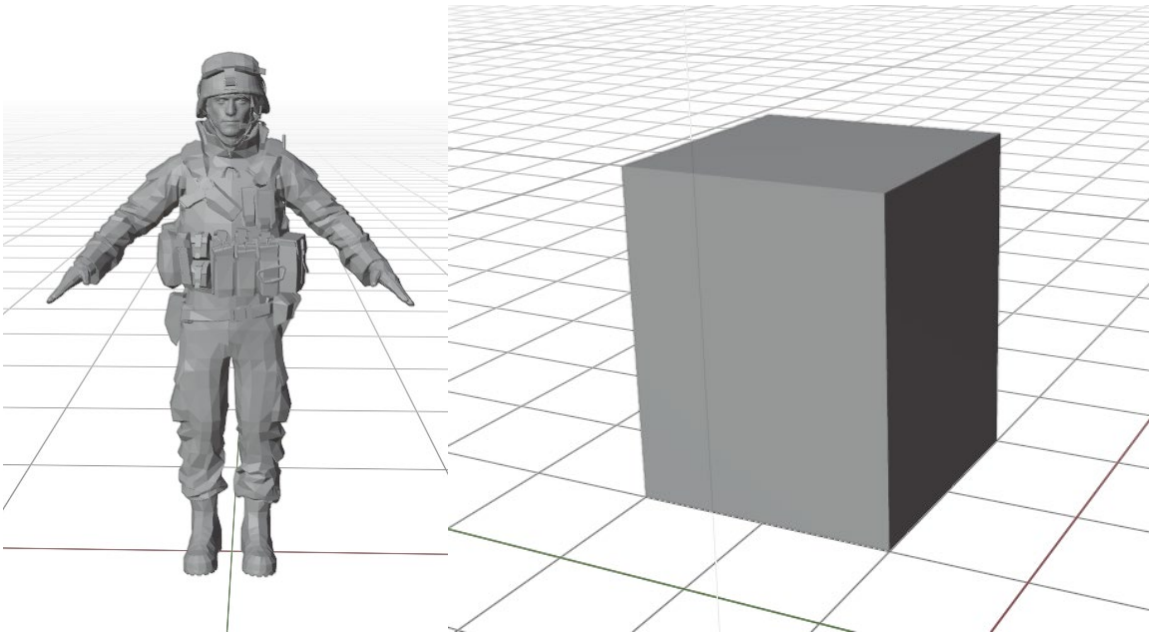


Figure 11. Open-field geometry model and box obstacle model shown in Blender.

A large quantity of simulation scenarios was run for this model with variations in charge location, charge mass, and obstacle location. Utilizing the existence of the block obstacle, a total of five different models for this soldier geometry was created. Exact scenarios will be further covered in Section 3.3, Computational Run Scenarios, and obstacle model variations will be shown in Section 3.3.2, Obstacles.

3.2.2 Sniper Geometry

To add some more realism to these simulations, a model of a laying down soldier firing a sniper rifle was designed. For this scenario, instead of an explosive bomb going off near the soldier, the blast is located at the tip of the sniper rifle to model a sniper rifle backfiring and exploding. The sniper geometry can be seen in Figure 12.

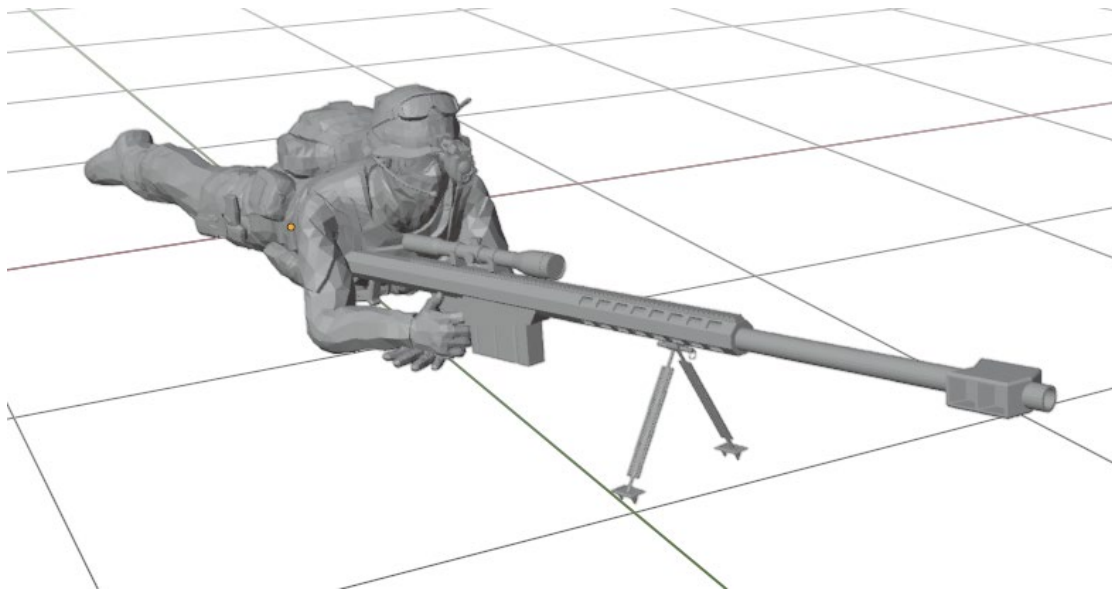


Figure 12. Sniper geometry model shown in Blender

The initial plan was to keep with the realism of this model, which would mean there wouldn't be much variation in regard to blast mass or location as its supposed to be a gun exploding. However, for the sake of mass data collection for ML training, we also created simulation scenarios with randomized blast location and mass around the sniper. This will be further covered in Section 3.3, Computational Run Scenarios.

3.2.3 Breacher Geometry

Another model that was created was the Breacher geometry which is a collection of eight soldiers in a line. This geometry was trying to model the scenario of a group of soldiers moving forwards together in close quarters to try and minimize exposure and damage from the front. This model can be seen in Figure 13.

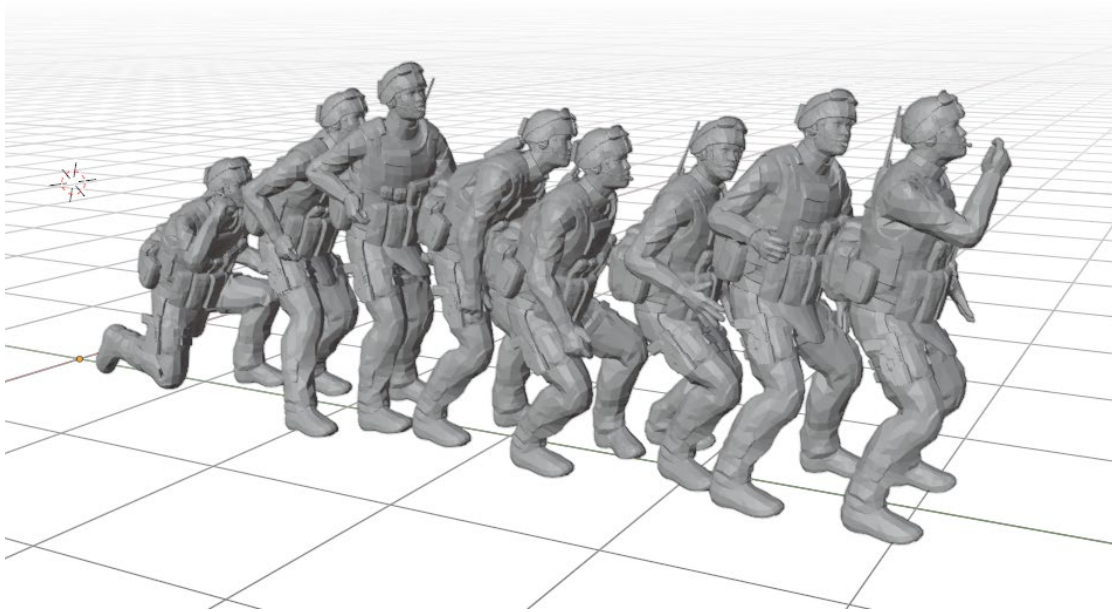


Figure 13. Breacher geometry model shown in Blender.

Since this was our largest model and had eight different sets of sensors for each soldier, we decided to run lots of different scenarios for it. Variations with blast location, blast mass, wall obstacles, as well as the addition of a riot shield for the first soldier were created. With the existence of a wall obstacle and riot shield, four different models were created for this soldier geometry. These obstacle model variations will be covered in Section 3.3.2, Obstacles.

3.2.4 Crouched Geometry

The last model that was created was the crouched geometry which is a single soldier. This geometry aimed to model a soldier crouching down and putting their hands over their head for protection from an incoming explosive. As this model represents a very likely scenario for a soldier in combat with an explosive going off, this was a welcomed addition for simulations. This geometry can be seen in Figure 14 below.

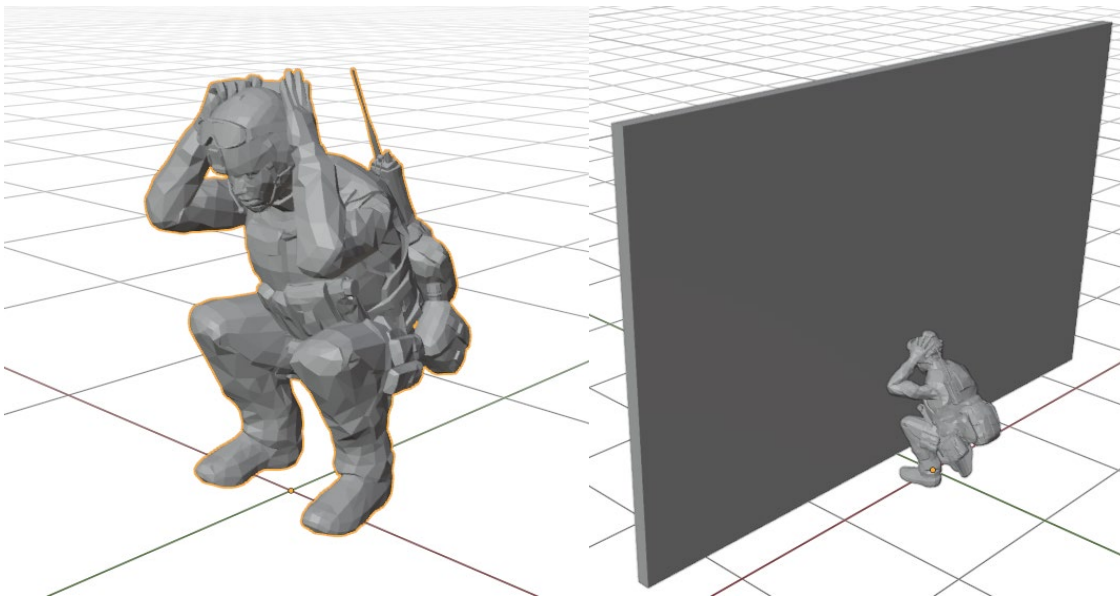


Figure 14. Crouched geometry model and wall obstacle shown in blender.

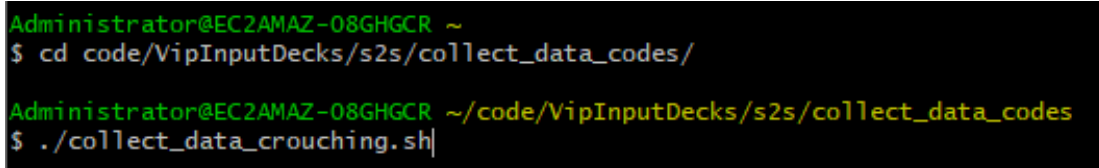
Due to the realism of this geometry, this model was also utilized with a breadth of simulation scenarios. The charge location and mass were varied greatly, and the addition of a wall obstacle was created. The idea for the wall geometries orientation was that in this model scenario, the soldier was crouching up against a wall to try and protect themselves from a blast behind them. With the addition of the wall, there were two different models created for this soldier geometry.

3.3 Computational Run Scenarios

Utilizing Viper, we ran a total of 2400 simulation scenarios. The idea behind the breakdown was to run an even 200 scenarios for each model created. This can be broken down into 800 for the breacher geometry (four models), 1000 for the open-field geometry (five models), 200 for the sniper geometry (one model), and 400 for the crouched geometry (two models). Computational run scenarios for all geometries simultaneously varied explosive mass and location which was done through the Monte Carlo feature. An important tool we utilized for creating a data library to store all these run simulation scenarios was Amazon's S3 Bucket.

3.3.1 Amazon S3 Bucket Transfer Code

The Amazon S3 Bucket is a great tool to use for compiling large data libraries for ease of access. Due to the sheer number of simulations, i.e., data files, that this research calls for, we decided to make use of this AWS service. In order actually move our data to the S3 Bucket, a series of steps needed to be taken. As discussed in Section 3.1.2, a small code was created to extract and compile the overpressure files created by Viper. In order to run this code and then send the compiled data to the Amazon S3 Bucket, certain commands need to be inputted into Cygwin64 Terminal, a Windows command application. These commands can be seen in Figure 15 and 16 below.



```
Administrator@EC2AMAZ-08GHGCR ~  
$ cd code/VipInputDecks/s2s/collect_data_codes/  
  
Administrator@EC2AMAZ-08GHGCR ~/code/VipInputDecks/s2s/collect_data_codes  
$ ./collect_data_crouching.sh|
```

Figure 15: Cygwin64 Terminal commands for running the File Collection Code.

The above two lines of command in Figure 18 show the process of locating a certain directory (which houses the compilation code file) on the computer using the command `cd` and then simply running the code by utilizing the command `./` followed by the name of the code (An example collection code can be seen in Figure 10). One thing to note is the file collection code is unique and needs to be changed depending on the situation, therefore, there were 12 different codes created for all 12 model variations. For ease of this code and ease of running through data in the machine learning code, all files were inputted into the same folder on the S3 Bucket, but we did separate them on the computer directory for our own reference.

```
Administrator@EC2AMAZ-08GHGCR ~
$ cd code/VipInputDecks/s2s/COLLECTED_DATA/

Administrator@EC2AMAZ-08GHGCR ~/code/VipInputDecks/s2s/COLLECTED_DATA
$ aws s3 cp --recursive ./OVPF_crouching s3://blast-sim-data/SCENARIO_TRAINER_THESIS
upload: OVPF_crouching\viper3d_th_overpressure_crouching_s10.txt to s3://blast-sim-data/SCENARIO_TRAINER_THESIS/viper3d_th_overpressure_crouching_s10.txt
upload: OVPF_crouching\viper3d_th_overpressure_crouching_s100.txt to s3://blast-sim-data/SCENARIO_TRAINER_THESIS/viper3d_th_overpressure_crouching_s100.txt
upload: OVPF_crouching\viper3d_th_overpressure_crouching_s106.txt to s3://blast-sim-data/SCENARIO_TRAINER_THESIS/viper3d_th_overpressure_crouching_s106.txt
upload: OVPF_crouching\viper3d_th_overpressure_crouching_s101.txt to s3://blast-sim-data/SCENARIO_TRAINER_THESIS/viper3d_th_overpressure_crouching_s101.txt
```

Figure 16: Cygwin64 Terminal commands for exporting files to AWS S3 Bucket

The single command line shown above in Figure 16 (first `cd` command line just finds directory as already stated, which in this case is different from previous directory) is the most important command line for this process. Utilizing the imported command `aws`, we are able to send all the overpressure files collected straight to the AWS S3 Bucket. This concludes the process of extraction, collection, and transfer of the overpressure files.

3.3.2 Obstacles

To increase variation and realism of data, we added obstacles surrounding the breacher, crouched, and open-field models. Since the obstacle created for the crouched geometry has already been covered and shown in Section 3.2.4, it will be excluded from this section. The focus for this section will be to show the obstacle variations for the breacher and open-field models as they have the most sub-models. For the open-field geometry, a box obstacle was included into the blast area, as already mentioned in Section 3.2.1. This obstacle was positioned in four different locations all evenly spaced 2 m in the x and 2 m in the y away from the soldier. This was the basis for the first four models of the open-field geometry. The fifth model was simply the soldier geometry surrounded by four box obstacles in all four locations to simulate a small two way corridor. All four obstacle locations can be seen in Figure 17.

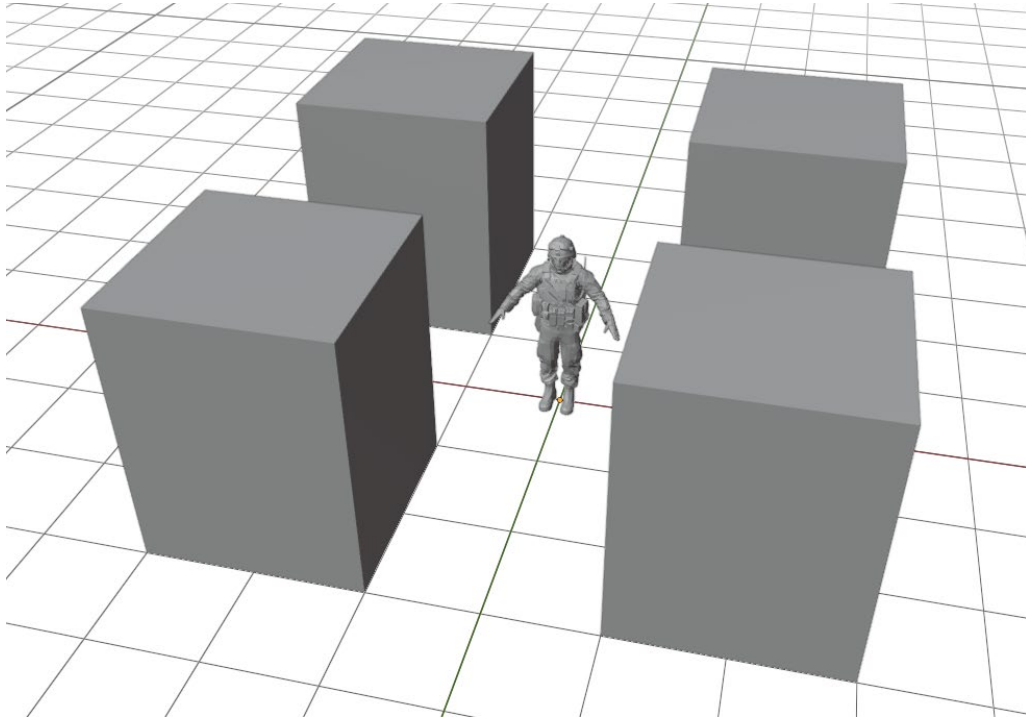


Figure 17: Isotropic view of Open-Field soldier geometry for clear view of the box obstacle and its four distinct locations in blender.

Since viper does not support textures or compositions of objects, the box was set to be an ideal and perfectly reflective object, which holds true with all objects/obstacles used in this research. Obviously, this is not realistic, but will still give us good data to train a machine learning algorithm with.

Furthermore, two different obstacles were created for the breacher geometry, as previously mentioned, resulting in a total of four different breacher models to run scenarios on (one with none, one with both). In one instance, a riot shield type obstacle was added to the front soldier of the breacher line. The other obstacle we decided to add was a two sided wall surrounding the breacher line. These obstacles can be seen in Figure 18.

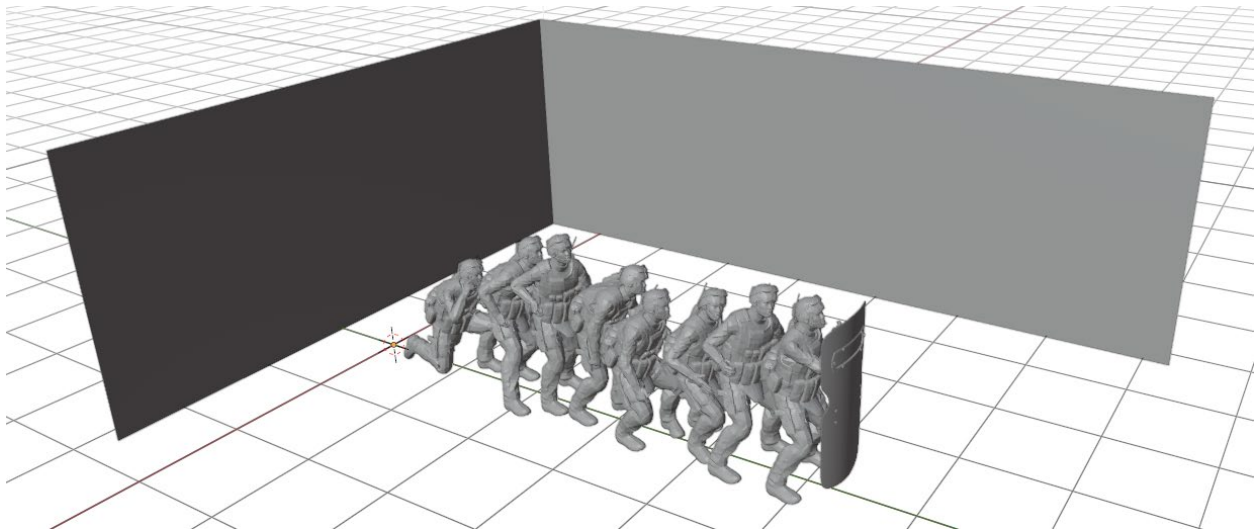


Figure 18: Breacher geometry model with the additional riot shield obstacle.

The only model that an obstacle was not added to was the sniper geometry. The reason for this was because the sniper model was based upon a very specific scenario of a bullet exploding. Since the blast here is on a smaller scale than the other model scenarios, the addition of obstacles wasn't of importance.

3.3.3 Blast Variation of Location and Magnitude for Scenarios

As mentioned in Section 3.1.2, two inputs for the explosive, mass, and location, were varied simultaneously to create a large quantity of unique blast scenarios. The software Viper seemed unable to handle any explosive mass close to or above 15 kg and would crash when initiated. To avoid this, all explosive masses for all models were generally set to be in a randomized range from 0-14 kg. The other variable was the blast location, which was varied in the x, y, and z directions. For all models, the z was randomized to be between 0-6 m and the x and y were varied differently for each case. The x and y location of the blast was randomized to be within the domain, which slightly differed from model to model. For example, the breacher geometry generally had a domain of -6 to 6 m in the x direction and -12 to 3 m in the y direction. However, the crouched geometry, since it was centered, had a domain of -6 to 6 m in the x and y directions. Small variations of these domains were made to fit better with their corresponding sub-models. An image of a breacher model scenario mid-simulation can be seen in Figure 19.

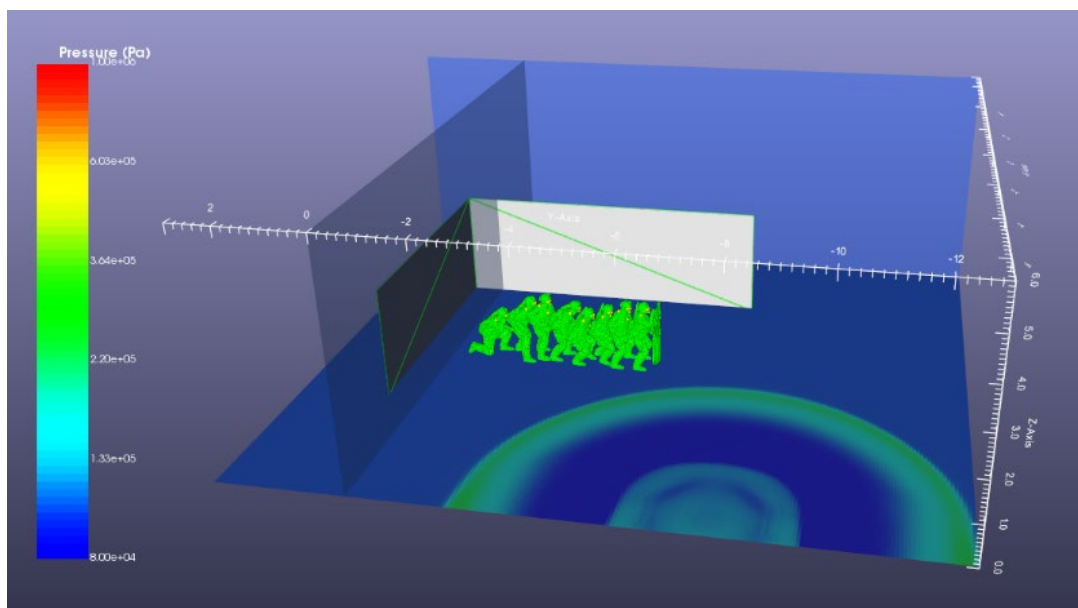


Figure 19. Viper blast simulation scenario for the breacher geometry with a double wall and shield obstacle showcasing the pressure waves from the blast.

3.4 Machine Learning Code

The most important portion of this thesis is the machine learning algorithm developed to be trained with the extracted Viper overpressure data. Utilizing three different popular machine learning algorithms, linear regression, ridge regression, and a deep neural network model known as the Keras Regressor, we were able to assess the applicability of each regarding this research. All three of these algorithms are readily available in an open-source python library for machine learning models called scikit-learn. According to Jason Brownlee, the author of the book Machine Learning Mastery with Python, these three machine learning models are previously known to be quite effective in predicting continuously varying quantities, which in this case is the peak pressure at the facial sensor locations [21].

To train and evaluate the validity of these three ML models for this thesis, the 2400 overpressure files were split into two data sets, a training set, and a validation set. The machine learning algorithms were first trained with the training data set at all thirteen sensor locations to give it accurate prediction power of facial overpressures. Then the validation set was sent through the code to evaluate the efficiency of this prediction by matching the predictions with the simulated facial overpressure from the validity set files.

The entire Python code can be seen in Appendix A of this thesis; however, this section will go over some important aspects of the code for better comprehension. To start out, a short yet important code is written out to define and read in the overpressure files from the S3 Bucket to be manipulated for the ML algorithms. This can be seen in Figure 20.


```

# Define first set of data to read
dataset = "SCENARIO_TRAINER_THESIS/"
root_folder1 = os.path.join(root, dataset)
print('Dataset Location = ',root_folder1)
files = fs.ls(root_folder1);
nfiles = len(files);

#import glob
#files1 = glob.glob("s3://blast-sim-data/breachers/*.txt");
files1 = files[1:nfiles];
files1 = files[1:2400];
files1[:nfiles]

```

Figure 20. Lines of code to define the dataset from the s3 Bucket and read in the overpressure files.

After all files have been read in, four important variables are defined. The first is **nscenarios** which is set to be equal to the number of files read in, i.e., 2400. Two sensor count variables are defined **nspp** and **nsw** which are set to be equal to the number of sensors per person in simulations (13) and the number of sensors worn by people in real life (3 – chest, shoulder, helmet), respectively. The final important variable defined is **PeakDurationExtraction** which, as its name states, is the duration of which pressure is known to be at its peak in the simulation and set to be 0.5.

To begin the process of manipulating these overpressure files to extract the peak pressures, the function **create_df** was defined to create a matrix of scenarios from the **root_folder**. This function is the basis for all manipulation and extraction of the overpressure files through a series of for loops. In order to make this code applicable to all scenarios, especially the ones with multiple soldiers compared to just one soldier, two lines of code were created to define the shape of **df** and create the variable **npeople**. This allowed the code to loop through extraction for loops by the number of people there were in the simulation model. This section of code can be seen below in Figure 21.

```

row, col = df.shape
#print(row, col)
npeople.append(int(col/nspp))
#print(npeople[ScenarioCounter])

```

Figure 21. Code to define the number of people in the simulation model.

One of the most important for loops in this code is the computation and extraction of the peak overpressures, which is the varying quantity we want to evaluate with the ML algorithms. By considering the number of people, the number of scenarios, and the number of sensors, this for loop creates a matrix of **MaxPressureValues** through the **append** command. This for loop can be seen in Figure 22.

```

print('npeople in Scenario ',ScenarioCounter,' = ',npeople[ScenarioCounter])
# FIRST COMPUTE MAX PRESSURES
for j in range(0,int(npeople[ScenarioCounter])):
    tmp = []
    for k in range (0,nspp):
        # Compute the max pressure values for this column,store only the pressures for this person
        tmp.append(df.iloc[:,j*nspp+k].max())
    # Add pressures from this person to the total pressure List
    MaxPressureValues.append(tmp)

```

Figure 22. For loop Python code that extracts the peak overpressure values from Viper simulation files.

The next step in the process was to define what each column was in **df** and assign a name to them. Each column represents the overpressure data from each sensor, so the three relevant sensors to BlackBox Biometrics – chest, one shoulder, helmet – were defined with names while the rest were just given numbers. This allowed for the analysis of the predictive power of all other sensor locations with regard to these three. In other words, the overpressure data from just the chest, one shoulder, and the back of the helmet will be used to predict the overpressures at all other locations, since these are the three that Black Box Biometrics use. This for loop can be seen in Figure 23.

```

for k in range (0,nspp):
    #print(str(col)+'P')
    # these numbers 0 1 2 may need to be modified based on nsw
    if k == 0:
        NewColumnNames.append('Chest-D')
    elif k == 1:
        NewColumnNames.append('Shoulder-D')
    elif k == 2:
        NewColumnNames.append('Helmet-D')
    elif k > nsw-1:
        NewColumnNames.append('Sensor-'+str(k)+'-D')
MaxDurationValuesAll.columns=NewColumnNames

```

Figure 23. For loop to define names for sensor overpressure data from Viper simulation database.

Once relevant data has been extracted and defined properly from the Viper overpressure files, the **create_df** function is executed and a series of data sets are created to store all peak pressure information such as maximum peak pressure and maximum peak duration of pressure. This execution code can be seen in Figure 24.

```

#df1, peak_df1, max_values_df1, max_durations_df1 = create_df (files1,root_folder1)
AllPeakInformation, MaxPressureValuesAll, MaxDurationValuesAll = create_df (files1,root_folder1)
#print(AllPeakInformation)

hf = h5py.File('ProcessedData.h5', 'w')
hf.create_dataset('AllPeakInformation', data=AllPeakInformation)
hf.create_dataset('MaxPressureValuesAll', data=MaxPressureValuesAll)
hf.create_dataset('MaxDurationValuesAll', data=MaxDurationValuesAll)
hf.close()

```

Figure 24. Creation of three data sets containing a series of peak overpressure information by execution of the **create_df** function.

As mentioned earlier, the 2400 scenarios were randomly split into two different data sets: a training set and a validation set. In order for the machine learning algorithm to properly do this, it was informed that the target variables for predicting facial overpressures were the chest, shoulder, and helmet. The predictor variables for predicting facial overpressures (basically aids to the ‘target’ variables) were the rest of the facial sensors, as seen in Figure 25.

```
# Regression Models for Pressure
PredictorColumnNames = MaxPressureValuesAll.columns[0:nsw]
TargetColumnNames = MaxPressureValuesAll.columns[nsw:nspp]
```

Figure 25. Python code to define target and predictor variables for the ML algorithms.

These two sets of variables (which are columns of a matrix) are then stored into data sets X and Y. These sets are split randomly into the training and validation data sets with a 67% to 33% split, respectively. This split can be seen in the following Python code in Figure 26.

```
X = AllPeakInformation[PredictorColumnNames]
y = AllPeakInformation[TargetColumnNames[j]]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=40)
# save the y testing data to compare the predictions to
SaveYTest = np.append(SaveYTest, y_test)
```

Figure 26. Python code to split the overpressure data library into training and validation sets randomly.

To measure the performance of the machine learning algorithms numerically, the coefficient of determination (R^2) was utilized between the prediction of the target variables from the training set directly from the Viper simulations and the prediction of the target variables from the validation set for the actual algorithms. This allowed us to quantitatively decipher our results and see how effective the different algorithms were at prediction. The R^2 values can range from 0 to 1 with 1 indicating that the algorithms perfectly predict the varying target quantities.

Chapter 4

Results

Viper not only outputs overpressure data files, but also gives a real time overpressure vs time data while the simulation is running. This showcases all thirteen sensors and their respective overpressures as the simulation runs through milliseconds. For reference and visual aid through what we are actually analyzing through the ML code, an example of this overpressure vs time graph can be seen in Figure 27 below.

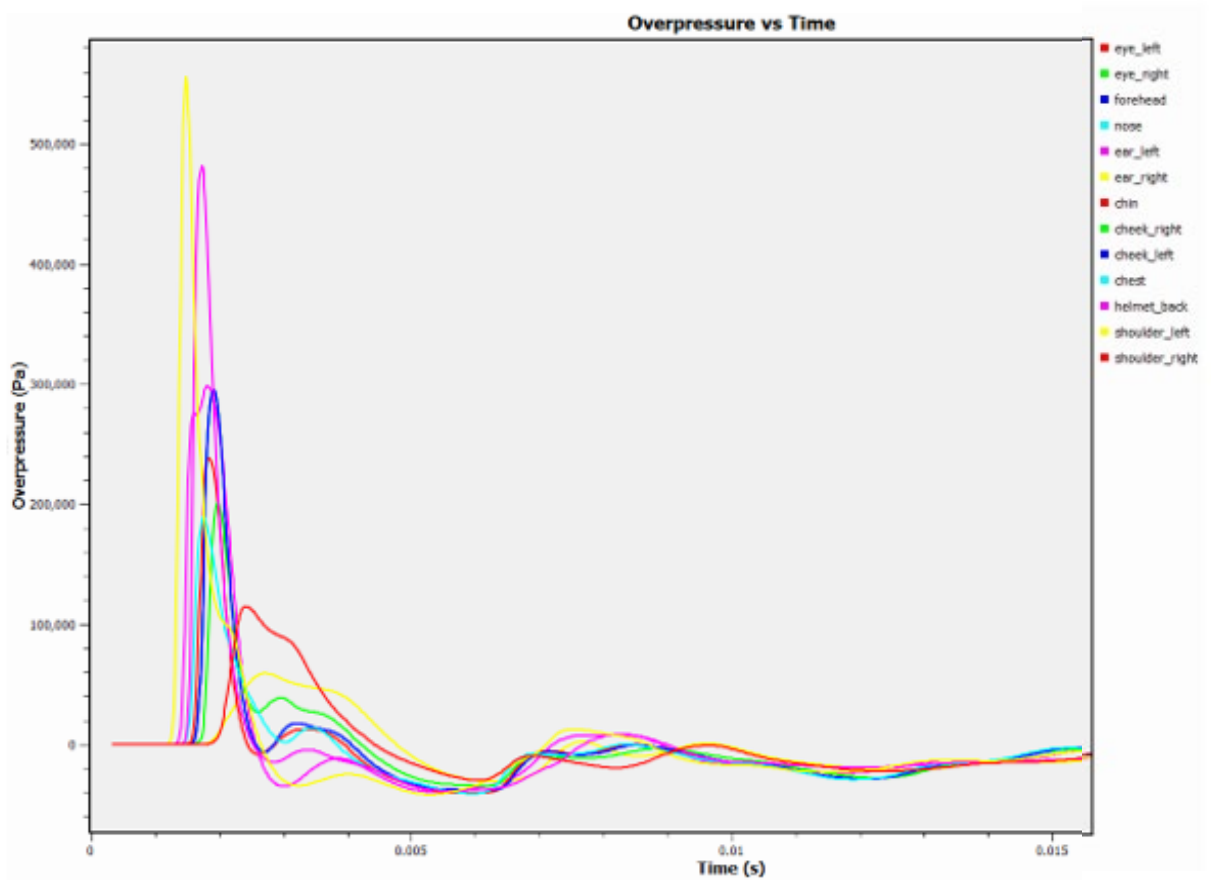


Figure 27. Viper Blast example Overpressure vs Time graph showcasing all thirteen sensors' pressures in real time.

This data was then processed and ran through the three Machine learning algorithms as previously mentioned. To start out, an initial test of just the linear and ridge regression

algorithms were ran with no output of R^2 to test feasibility and initial effectiveness. Only the peak pressure data extracted from the overpressure files was analyzed for predictive power. Additionally, only half the sensors were analyzed for prediction in order to save computational cost and time, since most come in sets of two anyway (i.e., left eye, right eye). All other graphs outputted straight horizontal lines as they were not tested. These initial tests can be seen in Figure 28.

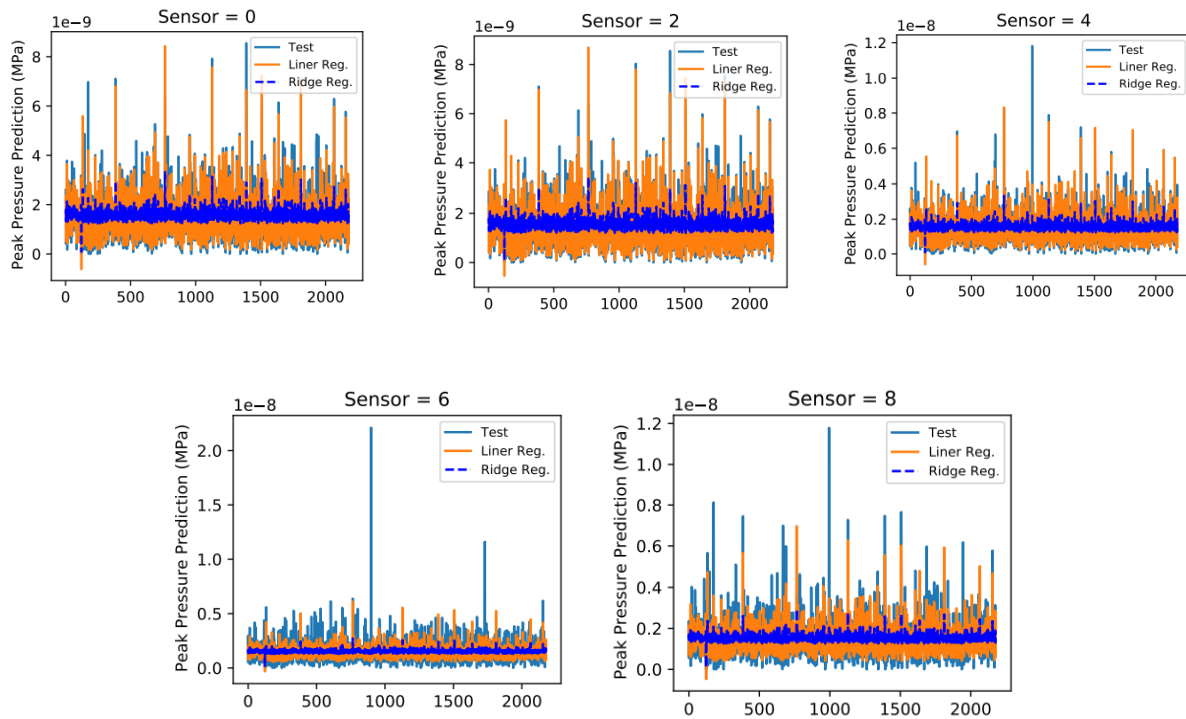
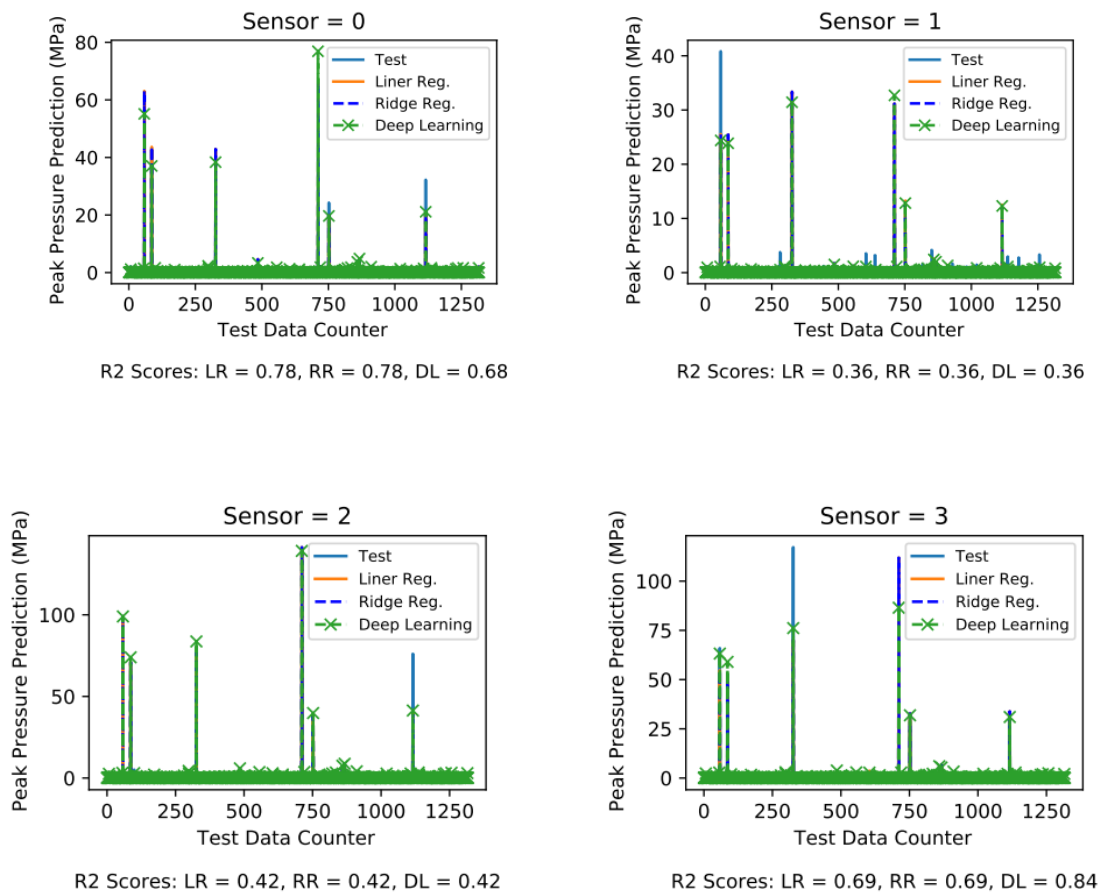


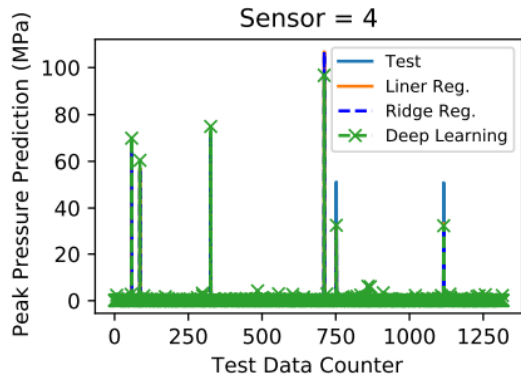
Figure 28. Series of peak overpressure prediction vs scenario count plots testing the feasibility of the linear and ridge regression ML models.

The sensors above correlate to left eye, forehead, right ear, chin, and right cheek labelled as sensors 0 through 8. These graphs quantify the predictive power of the targeted sensors – chest, shoulder, and helmet– with regard to the other facial sensors. As seen in the above graphs, the linear regression algorithm overall performed quite well on all sensors at lower peak pressures. Sensor 0 and sensor 2 both performed at higher peak pressures as well, but the other

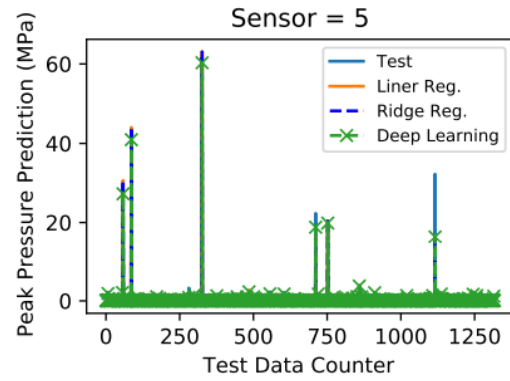
sensors seemed to miss the high spike peak pressures of some scenarios for the linear regression algorithm. The ridge regression model performed a lot worse and seemed to be off peak pressures by a large margin. One thing to note about this is that the ridge regression model did predict changes and spikes of peak pressure, just not at the right scale, which could be due to initial ridge regression code error.

Once the initial test was done and several rounds of debugging, we then added in the deep learning regression model (Keras Regression) to the python code and had it run through the overpressure data at a more precise scope for all 10 non-targeted sensors. This was run through in batches of 500 scenarios as running all 2400 at once was extremely time consuming and didn't allow for troubleshooting or cleaning of data. All 10 peak output plots can be seen in Figure 29.

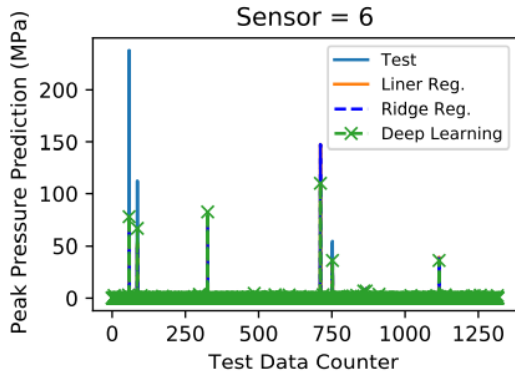




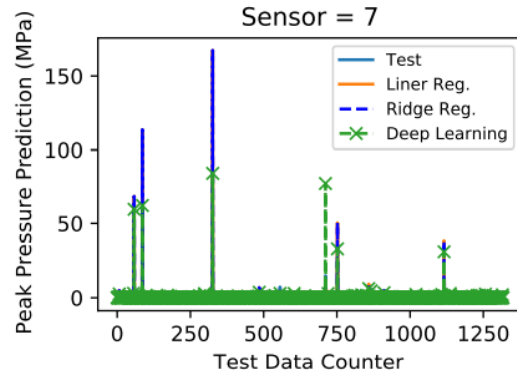
R2 Scores: LR = 0.70, RR = 0.70, DL = 0.76



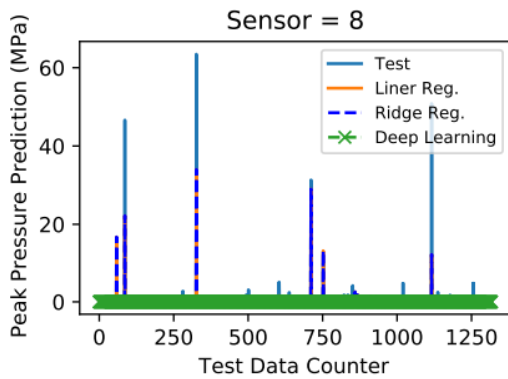
R2 Scores: LR = 0.82, RR = 0.82, DL = 0.85



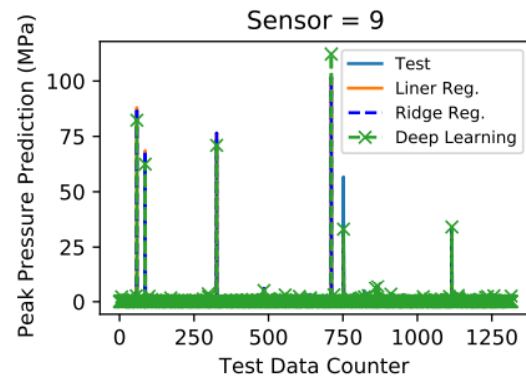
R2 Scores: LR = 0.36, RR = 0.36, DL = 0.51



R2 Scores: LR = -19.14, RR = -19.14, DL = -6.50



R2 Scores: LR = 0.68, RR = 0.68, DL = -0.01



R2 Scores: LR = -0.19, RR = -0.19, DL = -0.12

Figure 29. Output peak pressure prediction vs scenario count graph for all 10 non-targeted sensors for linear, ridge, and Keras regression ML algorithms.

Sensor 0, which refers to the left eye, exhibited a R^2 value of **0.78** for the linear and ridge regression models and **0.68** for the deep learning model. It can be inferred that at a smaller scope like this the ridge model increases in predictive power greatly to the point of matching the linear model. The deep learning model did not perform quite as well. Sensor 1, which refers to the right eye, outputted a R^2 value of **0.36** for all three models. This indicated an unnatural disagreement between the right and left eyes which seemed to be presented throughout the data.

Sensor 2, which refers to the forehead, gave a R^2 value of **0.42** for all three models. Sensor 3, which refers to the nose, exhibited a R^2 value of **0.69** for the linear and ridge regression models and **0.84** for the deep learning model. The nose, which previously has been mentioned as an important sensor location, gives an incredibly high deep learning model R^2 value, which is promising. Sensor 4, which refers to the right ear, showed a R^2 value of **0.70** for the linear and ridge and a **0.76** for the deep learning. Sensor 5, which refers to the left ear, exhibited a R^2 value of **0.82** for both the linear and ridge regression and **0.85** for the Keras regression. As previously seen, there was an unforeseen miscorrelation between the left and right sets of sensors, this one was less severe, though, and both are quite accurate in prediction.

Sensor 6, which refers to the chin, calculated a R^2 value of **0.36** for the linear and ridge models and **0.51** for the deep learning model. Sensor 7, which refers to the right cheek, outputted a R^2 value of **-19.14** for both the linear and ridge algorithms while the deep learning algorithm had a R^2 value of **-6.50**. This was the first extremely off case, which can only be accounted for through computational error. Sensor 8, which refers to the left cheek, showed a value of **0.68** for the linear and ridge regressions and **-0.01** for the deep learning. The last sensor, sensor 9, which refers to the right shoulder, was another outlier with values of **-0.19** for the linear and ridge and **-0.12** for the Keras.

Chapter 5

Discussion

The results make it clear that the ridge and linear regression models both perform exactly the same for this research, which did not agree with our initial testing. This can be explained by the fact that ridge regression is just a slight augmentation of linear regression as linear is extremely basic and simply ‘fits’ the best straight line to the datum. The purpose for ridge or other linear regression variations is to make it less susceptible to outliers or overfitting. Since the data library used in this research is extremely extensive and similar to each other, there are not that many outliers, which explains why these two regression models tend to each other in this research. Table 2 below showcases all R^2 values for all three machine learning algorithms for more concise access.

Table 2. R^2 values for linear, ridge, and deep learning Keras regression models for all 10 non-targeted sensor locations.

Sensor Location	Linear Regression	Ridge Regression	Deep Learning Keras
Left eye	0.78	0.78	0.68
Right eye	0.36	0.36	0.36
Forehead	0.42	0.42	0.42
Nose	0.69	0.69	0.84
Right ear	0.70	0.70	0.76
Left ear	0.82	0.82	0.85
Chin	0.36	0.36	0.51
Right cheek	-19.14	-19.14	-6.50
Left cheek	0.68	0.68	-0.01
Right shoulder	-0.19	-0.19	-0.12

When comparing the three ML models, it can be seen that the linear and ridge regression both performed better than the Keras regression on the left eye and left cheek only, while in all other locations the deep learning Keras model performed the same or better. The entire reason we added the Keras regression model because it performs continuously better with large increases in data and isn't such an 'average' fit as the linear and ridge are. With such a large quantity of overpressure files running through these algorithms, it makes sense that the deep learning model performs the best for the majority of locations. Additionally, the R^2 value of 0.84 for the deep learning model on the nose is extremely promising as the nose is quite important in datum locations for computational brain analysis.

One trend to note about the resulting R^2 values is that in the cases of ears, cheeks, and eyes, the left locations have much higher values. An interesting factor, and one to be tested upon more in the future, is the fact that for this thesis, the left shoulder was the shoulder chosen as the targeted shoulder wearing a blast gauge, and the right shoulder was left to be a training sensor. We believe that since the overpressure data from the left shoulder was used to predict facial overpressures and not the right, the sensor locations that were closer to the left shoulder were more accurate due to distance and reported higher R^2 values.

Another trend that needs to be addressed is the results of the bottom three sensor locations in Table 2, the right cheek, left cheek and right shoulder. All three display negative R^2 output values which can only be explained through computational error. This glitch can be credited to either an error in exact sensor node location (maybe inside of soldier's cheek, for example) or an error in the implementation and processing of the data in the python code for those locations. This could either be in the manipulation of the data at the beginning of the code or the actual processing through the algorithms.

Chapter 6

Conclusion

Overall, it is clear that machine learning has extreme potential in prediction of facial overpressures to be used in computational biomechanics research, and this thesis reiterates that. The python code developed in this research that implements a linear regression, ridge regression, and deep learning regression model, shows good promise for use with BlackBox Biometrics three sensor gauge system. Unfortunately, the results of this thesis are not yet where they should be in order for this company to actually make full use of this approach. Although the R^2 values for the non-targeted sensor locations show great qualities and tell us a lot about the next steps to take, they are a bit lower than expected and hoped for. The additions in this research and the resulting code are extremely helpful and a big step from previous work, however, there is still a lot of work to be done in the future.

Chapter 7

Future Work

Due to the scope and additive nature of this research, there is still a large amount of work to be done to increase its accuracy. This research is already years in the works, and can potentially go on for many more. For the purposes of this thesis, I only worked with 4 different soldier geometries with a total of 12 distinct models. The total count of overpressure file scenarios used to train the machine learning algorithms was 2400. Hypothetically, the accuracy of the algorithms predictive power for real life scenarios will increase as you increase the depth of simulation scenarios covered, and this accuracy can be continuously improved upon. Although the ML algorithms trained in this thesis could be used by BlackBox Biometrics to predict nose overpressure based upon their head, shoulder, and chest overpressures with decent accuracy, there is a lot of room for improvement. Moving forward, more soldier geometries and sub-models with obstacles needs to be designed as well as a larger and more varied datum library of scenarios. In theory, there is an infinite number of scenarios and geometries you could cover to increase accuracy, so for future work, the breadth of datum needs be increased a lot. Additionally, only three machine learning algorithms were trained for this thesis. Testing and training a large variation of machine learning algorithms could provide for more predictive power with certain ones. More in depth research on what kind of ML algorithm would work best for this research would have to be conducted. Ultimately, the reason for this research and thesis was to provide a fully trained machine learning algorithm for BlackBox Biometrics to use on their real overpressure datum, so the next big step would be to actually test these trained algorithms on their data. We believe that the code developed for this thesis isn't at that stage yet and still needs to be tweaked and debugged further as well as potentially added to.

Appendix A

Python Code for Machine Learning Algorithms

```
import s3fs
import matplotlib as plt
import numpy as np
import math
import pandas as pd
import os
import h5py

#Define root path
fs = s3fs.S3FileSystem()
root = "s3://blast-sim-data/"

# Define first set of data to read
dataset = "SCENARIO_TRAINER_THESIS/"
root_folder1 = os.path.join(root, dataset)
print('Dataset Location = ',root_folder1)
files = fs.ls(root_folder1);
nfiles = len(files);

#import glob
#files1 = glob.glob("s3://blast-sim-data/breachers/*.txt");
files1 = files[1:nfiles];
files1 = files[1:2];
files1[:nfiles]

# Important Variables
# number of scenarios that will be processed. Each scenario should be
# a different file from viper.
nscenarios = files1
print('nscenarios = ',len(nscenarios))
# number of sensors per person
nspp = 13
# Number of Sensors Worn (nsw) by person
nsw = 3 # chest, shoulder, helmet for B3
# Peak duration extraction (time for which pressure is above x% of max)
```

```

PeakDurationExtraction = 0.5
def create_df(scenarios,root_folder):
    all_max_values = []
    all_max_durations = []
    AllPeakInformation = []
    StartEndIndices= {}
    OverPressureDuration = []
    combinedNames = []
    NewColumnNames = []
    npeople = []
    OverPressureDurationAll = pd.DataFrame()
    MaxDurations = pd.DataFrame()
    MaxPressureValuesAll = pd.DataFrame()
    MaxDurationValuesAll = pd.DataFrame()
    combinedDf = pd.DataFrame()
    ScenarioCounter = 0

    # each file represents a different scenario. each scenario could have
    # multiple people (npeople) with with multiple tracer locations or
    # number of sensors per person (nspp)
    for scenario in scenarios:
        ScenarioFileName = os.path.basename(scenario)
        ScenarioFilePath = os.path.join(root_folder,ScenarioFileName)
        df = []
        df = pd.read_csv(ScenarioFilePath, delim_whitespace=True, header=None)
        # remove time (the first column)
        df = df.drop(df.columns[[0]], axis=1) # df.columns is zero-based pd.Index
        #print('df = ',df)
        # read only 1st column to store time trace
        time = pd.read_csv(ScenarioFilePath,delim_whitespace=True,usecols=[0], header=None)
        #convert array to list
        #time = time.values.tolist()
        time = np.array(time)

        row, col = df.shape
        #print(row, col)
        npeople.append(int(col/nspp))
        #print(npeople[ScenarioCounter])

    #print(df.shape)

```

```

#print(df)
# makes single array with chest thresholds for each person, each sensor
# basically half value of maximum in each column in df
thresholds=PeakDurationExtraction*df.max(axis=0)
# number of timepoints (number of rows in viper file)
#print(time)
ntimepoints=len(time)
#print('ntimepoints = ',ntimepoints)

# Initialize OverpressureDuration
OverPressureDuration = []
OverpressureDurationAll = []
MaxPressureValues = []
MaxDurationValues = []
frames = []

print('npeople in Scenario ',ScenarioCounter,' = ',npeople[ScenarioCounter])
# FIRST COMPUTE MAX PRESSURES
for j in range(0,int(npeople[ScenarioCounter])):
    tmp = []
    for k in range (0,nspp):
        # Compute the max pressure values for this column,store only the pressures for this
person
        tmp.append(df.iloc[:,j*nspp+k].max())
        # Add pressures from this person to the total pressure list
        MaxPressureValues.append(tmp)

#### NOW COMPUTE DURATIONS
for j in range(0,int(npeople[ScenarioCounter])):
    tmp = []
    for k in range (0,nspp):
        start_index = -1
        end_index = -1
        for i in range(0, ntimepoints):
            if (df.iloc[i,j*nspp+k]>thresholds.iloc[j*nspp +k] and start_index == -1):
                start_index=i
                #break
        for i in range(start_index, ntimepoints):
            if df.iloc[i,j*nspp+k]<thresholds.iloc[j*nspp +k] and end_index == -1:
                #print('min detected !','sensor ',k+1, ' i index = ',i)

```



```

        end_index=i
        #break
        #print('start_index = ', start_index)
        #print('end_index = ', end_index)
        StartEndIndices[j*nspp+k+0] = start_index
        StartEndIndices[j*nspp+k+1] = end_index
        tmp.append(time[end_index][0] - time[start_index][0])
        MaxDurationValues.append(tmp)
        #print('MaxDurationValues = ', MaxDurationValues)

    print('finished processing scenario ',ScenarioCounter,...')
    # save pressures into single array, first convert to dataframe
    MaxPressureValuesDF = pd.DataFrame(MaxPressureValues)
    MaxPressureValuesAll =
pd.concat([MaxPressureValuesAll,MaxPressureValuesDF],ignore_index=True)
    # save pressures into single array, first convert to dataframe
    MaxDurationValuesDF = pd.DataFrame(MaxDurationValues)
    MaxDurationValuesAll = pd.concat([MaxDurationValuesAll,
MaxDurationValuesDF],ignore_index=True)
    #print('MaxPressureValuesDF = ', MaxPressureValuesDF)
    ScenarioCounter = ScenarioCounter + 1
    #print('MaxPressureValuesAll = ', MaxPressureValuesAll)
    #print('MaxDurationValuesAll = ', MaxDurationValuesAll)

#create column names for pressure df
for k in range (0,nspp):
    # these numbers 0 1 2 may need to be modified based on nsw
    if k == 0:
        NewColumnNames.append('Chest-P')
    elif k == 1:
        NewColumnNames.append('Shoulder-P')
    elif k == 2:
        NewColumnNames.append('Helmet-P')
    elif k > nsw-1:
        NewColumnNames.append('Sensor-'+str(k)+'-P')
    MaxPressureValuesAll.columns=NewColumnNames
    NewColumnNames = []

for k in range (0,nspp):

```

```

#print(str(col)+'P')
# these numbers 0 1 2 may need to be modified based on nsw
if k == 0:
    NewColumnNames.append('Chest-D')
elif k == 1:
    NewColumnNames.append('Shoulder-D')
elif k == 2:
    NewColumnNames.append('Helmet-D')
elif k > nsw-1:
    NewColumnNames.append('Sensor-'+str(k)+'-D')
MaxDurationValuesAll.columns=NewColumnNames

AllPeakInformation=pd.concat([MaxPressureValuesAll, MaxDurationValuesAll], axis=1)
#print(AllPeakInformation)
l1 = MaxDurationValuesAll.columns
l2 = MaxPressureValuesAll.columns
#print('l1',l1)
#print('l2',l2)
colNames = zip(l2, l1)
#print(colNames)
combinedNames = [name for pair in colNames for name in pair]
AllPeakInformation = AllPeakInformation[combinedNames]
#print(AllPeakInformation)
return AllPeakInformation, MaxPressureValuesAll, MaxDurationValuesAll

#df1, peak_df1, max_values_df1, max_durations_df1 = create_df(files1,root_folder1)
AllPeakInformation, MaxPressureValuesAll, MaxDurationValuesAll = create_df
(files1,root_folder1)
#print(AllPeakInformation)

hf = h5py.File('ProcessedData.h5', 'w')
hf.create_dataset('AllPeakInformation', data=AllPeakInformation)
hf.create_dataset('MaxPressureValuesAll', data=MaxPressureValuesAll)
hf.create_dataset('MaxDurationValuesAll', data=MaxDurationValuesAll)
hf.close()

#Linear regression for peak pressure and peak duration (separately)
import sklearn
#from matplotlib import pyplot
import matplotlib.pyplot as plt

```

```

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import model_selection
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from math import sqrt
import pickle

# used for DL
import tensorflow as tf
import os
tf.get_logger().setLevel('WARNING')
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
print(tf.__version__)
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(3, input_dim=3, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# Regression Models for Pressure

```

```

PredictorColumnNames = MaxPressureValuesAll.columns[0:nsw]
TargetColumnNames = MaxPressureValuesAll.columns[nsw:nspp]
SaveYTest = []
SaveYPred_LinReg = []
SaveYPred_rr = []
SaveYPred_DL = []
NumTests = 0
#for j in range(0,nspp-nsw):
for j in range(0,2):
    print('Sensor: ',j)
    X = AllPeakInformation[PredictorColumnNames]
    y = AllPeakInformation[TargetColumnNames[j]]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=40)
    # save the y testing data to compare the predictions to
    SaveYTest = np.append(SaveYTest, y_test)

    # Set the number of tests...will be used for plotting.
    if j==0:
        NumTests = len(X_test)
    if j>0 and NumTests != len(X_test):
        print('Number of training tests are different! Will cause issues when plotting')

    # Linear Regression
    linreg = LinearRegression().fit(X_train, y_train)
    y_pred_LinReg = linreg.predict(X_test)
    SaveYPred_LinReg = np.append(SaveYPred_LinReg, y_pred_LinReg )
    print('Lin. Reg. MSE = %.2f % np.sqrt(mean_squared_error(y_test,y_pred_LinReg)))
    print('Lin. Reg. R2 = %.2f % r2_score(y_test, y_pred_LinReg))
    print(' ')

    # Ridge Regression
    rr = Ridge(alpha=0.01)
    rrf = rr.fit(X_train, y_train)
    y_pred_rr = rrf.predict(X_test)
    SaveYPred_rr = np.append(SaveYPred_rr, y_pred_rr )
    print('Ridge Reg. MSE = %.2f % np.sqrt(mean_squared_error(y_test,y_pred_rr)))
    print('Ridge Reg. R2 = %.2f % r2_score(y_test, y_pred_rr))
    print(' ')

    # Deep Learning - evaluate DL model

```

```

estimator = KerasRegressor(build_fn=baseline_model, epochs=100, batch_size=5, verbose=0)
kfold = KFold(n_splits=10)
results = cross_val_score(estimator, X, y, cv=kfold)
estimator.fit(X_train, y_train)
y_pred_dl= estimator.predict(X_test)
print('ML y_train Mean Squared Error: %.2f % np.sqrt(mean_squared_error(y_test,
y_pred_dl)))
print('ML y_train R2 score: %.2f % r2_score(y_test, y_pred_dl))
SaveYPred_DL = np.append(SaveYPred_DL, y_pred_dl)
print( '----- ')

```

```

print('Length of SaveYTest = ',len(SaveYTest))
print('Length of SaveYPred_rr = ',len(SaveYPred_rr))
print('Length of SaveYPred_LinReg = ',len(SaveYPred_LinReg))
print('Length of SaveYPred_DL = ',len(SaveYPred_DL))

```

```

from matplotlib.backends.backend_pdf import PdfPages
import matplotlib as mpl
mpl.rcParams.update(mpl.rcParamsDefault)

```

```

pdf_pages = PdfPages('Training-Results.pdf')
print('NumTests =',NumTests)
PlotPairs = int((nspp-nsw)/2)
#print(PlotPairs)
print
for j in range(0,PlotPairs):
    fig, axs = plt.subplots(1, 2,figsize=(10,5))
    iterator = (nsw-1)*j
    tmp_plot1 = []
    tmp_plot2 = []
    tmp_plot3 = []
    tmp_plot4 = []
    for i in range(0,NumTests):
        tmp_plot1 = np.append(tmp_plot1,SaveYTest[iterator*NumTests + i])
        tmp_plot2 = np.append(tmp_plot2,SaveYPred_LinReg[iterator*NumTests + i])
        tmp_plot3 = np.append(tmp_plot3,SaveYPred_rr[iterator*NumTests + i])
        tmp_plot4 = np.append(tmp_plot4,SaveYPred_DL[iterator*NumTests + i])
    axs[0].plot( tmp_plot1, label="Test" )
    axs[0].plot( tmp_plot2, label="Liner Reg." )

```

```

axs[0].plot( tmp_plot3,'--b' , label ="Ridge Reg.")
axs[0].plot( tmp_plot4,'--x' , label ="Deep Learning")
axs[0].set_xlabel("Test Data Counter")
axs[0].set_ylabel("Peak Pressure Prediction")
axs[0].set_title('Sensor = '+str((nsw-1)*j))
axs[0].legend(loc="upper right")
plt.tight_layout(pad=8.0)

```

```

iterator = (nsw-1)*j+1
tmp_plot1 = []
tmp_plot2 = []
tmp_plot3 = []
tmp_plot4 = []
for i in range(0,NumTests):
    tmp_plot1 = np.append(tmp_plot1,SaveYTest[iterator*NumTests + i])
    tmp_plot2 = np.append(tmp_plot2,SaveYPred_LinReg[iterator*NumTests + i])
    tmp_plot3 = np.append(tmp_plot3,SaveYPred_rr[iterator*NumTests + i])
    tmp_plot4 = np.append(tmp_plot4,SaveYPred_DL[iterator*NumTests + i])
axs[1].plot( tmp_plot1, label="Test" )
axs[1].plot( tmp_plot2, label ="Liner Reg." )
#axs[1].set_xlabel("Test Data Counter")
axs[1].set_ylabel("Peak Pressure Prediction")
axs[1].plot( tmp_plot3,'--b' , label ="Ridge Reg.")
axs[1].plot( tmp_plot4,'--x' , label ="Deep Learning")
axs[1].legend(loc="upper right")
axs[1].set_title('Sensor = '+str((nsw-1)*j+1))
t = "An item is found at i"
#axs[1].text(0, 0, t, wrap=True)
axs[1].set_xlabel(r"Test Data Counter"
                "\n"
                "hi3")
pdf_pages.savefig(fig)

```

```

pdf_pages.close()
print("Training Complete. Images saved.")

```

BIBLIOGRAPHY

- [1] C. C. Taylor, P.A. and Ford, “Simulation of Head Impact Leading to Traumatic Brain Injury,” *Int. NeuroTrauma Lett.*, no. October 2016, 2007, [Online]. Available: <http://isn-csm.mit.edu/literature/2007-ijnl-taylor.pdf>.
- [2] S. H. Pedersen, A. Lilja-Cyron, R. Astrand, and M. Juhler, “Monitoring and Measurement of Intracranial Pressure in Pediatric Head Trauma,” *Front. Neurol.*, vol. 10, no. January, pp. 1–9, 2020, doi: 10.3389/fneur.2019.01376.
- [3] Centers for Disease Control and Prevention, “TBI: Get the Facts | Concussion | Traumatic Brain Injury | CDC Injury Center,” *U.S. Department of Health & Human Services*. pp. 1–3, 2019, [Online]. Available: https://www.cdc.gov/traumaticbraininjury/get_the_facts.html.
- [4] P. Rønning, E. Helseth, N. O. Skaga, K. Stavem, and I. A. Langmoen, “The effect of ICP monitoring in severe traumatic brain injury: A propensity score–weighted and adjusted regression approach,” *J. Neurosurg.*, vol. 131, no. 6, pp. 1896–1904, 2019, doi: 10.3171/2018.7.JNS18270.
- [5] Johns Hopkins Medicine, “Sports Injury Statistics.” 1 - 4, 2021, [Online]. Available: <https://www.hopkinsmedicine.org/health/conditions-and-diseases/sports-injuries/sports-injury-statistics>.
- [6] D. Wallace and S. Rayner, “Combat helmets and blast traumatic brain injury,” *J. Mil. Veterans. Health*, vol. 20, no. 1, pp. 10–17, 2012.
- [7] K. Laksari, M. Kurt, H. Babaee, S. Kleiven, and D. Camarillo, “Mechanistic Insights into Human Brain Impact Dynamics through Modal Analysis,” *Phys. Rev. Lett.*, vol. 120, no. 13, p. 138101, 2018, doi: 10.1103/PhysRevLett.120.138101.

- [8] A. H. S. Holbourn, “Mechanics of Head Injuries,” *Lancet*, vol. 242, no. 6267, pp. 438–441, 1943, doi: 10.1016/S0140-6736(00)87453-X.
- [9] S. Heimbs, J. Ritzer, and J. Markmiller, “A Numerical Method for Blast Shock Wave Analysis of Missile Launch from Aircraft,” *Int. J. Aerosp. Eng.*, vol. 2015, no. May, 2015, doi: 10.1155/2015/897213.
- [10] V. Karlos, G. Solomos, and M. Larcher, “Analysis of the blast wave decay coefficient using the Kingery–Bulmash data,” *Int. J. Prot. Struct.*, vol. 7, no. 3, pp. 409–429, 2016, doi: 10.1177/2041419616659572.
- [11] P. A. Taylor and C. C. Ford, “Simulation of blast-induced early-time intracranial wave physics leading to traumatic brain injury,” *J. Biomech. Eng.*, vol. 131, no. 6, pp. 1–11, 2009, doi: 10.1115/1.3118765.
- [12] J. Iverson, “Glasgow Coma Scale.” StatPearls Publishing, Treasure Island, FL, 3–1, 2021, [Online]. Available: <https://www.ncbi.nlm.nih.gov/books/NBK513298/>.
- [13] A. S. Alali *et al.*, “Intracranial pressure monitoring in severe traumatic brain injury: Results from the american college of surgeons trauma quality improvement program,” *J. Neurotrauma*, vol. 30, no. 20, pp. 1737–1746, 2013, doi: 10.1089/neu.2012.2802.
- [14] P. H. Raboel, J. Bartek, M. Andresen, B. M. Bellander, and B. Romner, “Intracranial pressure monitoring: Invasive versus non-invasive methods-A review,” *Crit. Care Res. Pract.*, vol. 2012, pp. 3–7, 2012, doi: 10.1155/2012/950393.
- [15] “BlackBox Biometrics®, Inc.” Rochester, NY, 5–23, 2020 [Online]. Available: <https://b3inc.com/>.
- [16] Defense Advanced Research Projects Agency, “Blast gauge.” 2015, [Online]. Available: <https://www.darpa.mil/about-us/timeline/blast-gauge>.

- [17] L. Katch, “Reverse Source Localization for Identification of Overpressure Sources Based on Wearable Blast Gauges,” The Pennsylvania State University, 2021.
- [18] Q. Bi, K. E. Goodman, J. Kaminsky, and J. Lessler, “What is machine learning? A primer for the epidemiologist,” *Am. J. Epidemiol.*, vol. 188, no. 12, pp. 2222–2239, 2019, doi: 10.1093/aje/kwz189.
- [19] National Institute of Biomedical Imaging and Bioengineering, “Computational modeling,” *Chemical and Engineering News*, vol. 79, no. 35, p. 13, 2001, doi: 10.4135/9781412976626.n29.
- [20] S. Karim and T. R. Soomro, “What Is Cloud Computing?” Amazon, pp. 1–27, 2020, [Online]. Available: https://aws.amazon.com/what-is-cloud-computing/?nc2=h_ql_le_int_cc.
- [21] J. Brownlee, *Machine Learning Mastery with Python: understand your data, create accurate models, and work projects end-to-end*. Machine Learning Mastery, independently published, 2016.

ACADEMIC VITA

*Attached below

JACKSON MACKAY

jcm6102@psu.edu

EDUCATION

The Pennsylvania State University

College of the Engineering

Bachelor of Science in Mechanical Engineering

Relevant Coursework: Computational Tools, Circuit Analysis, Thermodynamics, Heat Transfer, Finite Element Engineering, Mechatronics, Mechanical Design, Fluid Flow, Material Science

University Park, Pennsylvania
Expected Graduation: May 2022

Schreyers Honors College

Accepted into the university's prestigious honors college. Currently enrolled in honors classes and working on a professional thesis to be submitted to the college before graduation. Held to the highest standards that the college upholds.

University Park, Pennsylvania
August 2018-Present

WORK EXPERIENCE

Critchfield Mechanical, Inc.

Project Engineer Intern

San Jose, California
May 2021-August 2021

- Helped in the process of bidding, estimating, and designing HVAC systems for buildings in Silicon Valley
- Calculated air distribution requirements for buildings based off of numerous criteria
- Sized and designed duct work, mechanical piping, and HVAC equipment to meet air distribution requirements.
- Prepared full cost estimates of materials, labor, fabrication, and design for HVAC systems

Computational Biomechanics Lab (Honors Thesis)

Undergraduate Researcher

State College, Pennsylvania
November 2020-Present

- Developing and printing 3D CAD models of wrestling head gear prototypes that contain head trauma sensors
- Making use of AWS computing to run thousands of blast overpressure simulation scenarios on soldiers
- Utilizing machine learning to predict facial overpressures for brain neural analysis through varied overpressure sim files

INVOLVEMENT

Engineers Without Borders

Active General Body Member/Volunteer

University Park, Pennsylvania
September 2018-2020

- Attending weekly meetings and volunteering hours to help on yearly EWB projects
- Designing and developing models and systems capable of sustaining the projects intentions
- Determining how to supply and distribute a clean water source to Namutamba, Uganda

Feeding the Community

Head Community Outreach Chair

University Park, Pennsylvania
October 2020-October 2021

- Attending meetings and volunteering hours to collect and provide resources for local food pantries
- Designing food drive boxes to be put out around the community to collect non-perishables
- Contacting food pantries to donate to as well as local businesses to use as donators or food drive locations

Sigma Pi

Executive Secretary and Alumni Relations Chair

University Park, Pennsylvania
January 2019-Present

- Promoting fellowship and developing leadership roles in the local community as well as Alumni connections
- Raising money for Penn State Dance Marathon (THON); largest student run philanthropy benefitting pediatric cancer
- Attending weekly chapter meeting as well as exec meetings while taking Secretary notes and providing leadership assistance

SKILLS/ HONORS/INTERESTS

Key Skills

CAD Capabilities (SolidWorks)

- Enrolled in multiple mechanical engineering courses in CAD and utilized it in an undergraduate research project

Coding Capabilities

- Studied MATLAB in multiple courses that integrated advanced calculus and tools as well as C++ through Arduino

Arduino Circuitry

- Analyzed circuits in multiple courses that utilized the Arduino circuit boards

Microsoft Office

- 10+ years of experience with Word, Excel, PowerPoint, and Outlook

Honors: Schreyers Honors Scholarship, Alpha Lambda Delta Honors Society Inductee, Dean's List for four semesters

Interests: Mechanical Engineering, CAD Design, Circuitry, Civil Engineering, Physics, Aerospace, Astronomy, Finance & Business