[Editorial comments appear in this style (italic blue font in brackets, hidden font) and will not normally print with your document. You may delete the editorial comments after you understand them.]

ECE/COE 1896

Senior Design

Manual Transmission Trainer Final Report

Prepared By: Benjamin Rarrick

Nick Albanese

Jimmy Mansmann

[This page is intentionally blank.]

# Table of Contents

*[When you want to update your table of contents (i.e. prior to turn-in), Right-click on table, select ‘update field’ and select ‘update entire table’]*

[Table of Contents i](#_Toc521090701)

[1. Executive Summary 1](#_Toc521090702)

[2. Problem Definition 1](#_Toc521090703)

[3. Background 2](#_Toc521090704)

[4. System Requirements 3](#_Toc521090705)

[4.1 System 3](#_Toc521090706)

[4.1.1 The system shall include an OBD II Scanner 3](#_Toc521090707)

[4.1.2 The system shall include a wireless communication setup to receive real time data 3](#_Toc521090708)

[4.1.3 The System shall implement a decision tree for determining current gear 4](#_Toc521090709)

[4.1.4 The system shall include a PC display 4](#_Toc521090710)

[4.2 OBD II Scanner 4](#_Toc521090711)

[4.2.1 The OBD II Scanner must send data in real time 4](#_Toc521090712)

[4.2.2 The OBD II scanner must rely only on power from OBD II port 4](#_Toc521090713)

[4.2.3 The system shall include an ODB II reading UART 4](#_Toc521090714)

[4.2.4 The system shall include Gyroscope/Accelerometer/Magnometer breakout 4](#_Toc521090715)

[4.2.5 The system shall include Bluetooth breakout 4](#_Toc521090716)

[4.2.6 The system shall include a Custom Designed MCU 5](#_Toc521090717)

[4.2.7 The whole system shall be housed in a custom design box 5](#_Toc521090718)

[4.3 Communication System 5](#_Toc521090719)

[4.3.1 The communication must be implemented using Bluetooth 5](#_Toc521090720)

[4.3.2 The communication must operate with a workable latency 5](#_Toc521090721)

[4.3.3 The communication must be reliable 5](#_Toc521090722)

[4.4 Data Processing and Algorithm Implementation 5](#_Toc521090723)

[4.4.1 The data from the OBD will be transmitted via Bluetooth and C++ program 5](#_Toc521090724)

[4.4.2 The data received by the Java Application will be parsed and processed 5](#_Toc521090725)

[4.4.3 The data that has been processed will be used to traverse the trained tree to find current gear 6](#_Toc521090726)

[4.4.4 The decision tree algorithm will be fast enough to not introduce additional latency 6](#_Toc521090727)

[4.5 System Display 6](#_Toc521090728)

[4.5.1 The display will be the user's laptop 6](#_Toc521090729)

[4.5.2 The display will indicate which gear the car is currently in 6](#_Toc521090730)

[4.5.3 The display will instruct the user on when to upshift and to which gear 6](#_Toc521090731)

[4.5.4 The display will contain a textbox to input the actual current gear of the driver 6](#_Toc521090732)

[5. Design Constraints 7](#_Toc521090733)

[5.1 The system will not distract the user from their driving responsibilities 7](#_Toc521090734)

[5.2 The system will require no external power supply 7](#_Toc521090735)

[5.3 The system will be real time 7](#_Toc521090736)

[5.4 The system will be compatible with just one vehicle 7](#_Toc521090737)

[6. Evaluation of Design Concepts 7](#_Toc521090738)

[6.1 Overall System 7](#_Toc521090739)

[6.2 OBD II Scanner 8](#_Toc521090740)

[6.2.1 OBD II Scanner Functional Description 8](#_Toc521090741)

[6.2.2 OBD II Scanner Housing Functional Description 10](#_Toc521090742)

[6.3 Connectivity Between Microprocessor and User Device 10](#_Toc521090743)

[6.3.1 Connectivity Between Microprocessor and User Device Functional Description 10](#_Toc521090744)

[6.3.2 Connectivity Between Microprocessor and User Device Design Option 1 10](#_Toc521090745)

[6.3.3 Connectivity Between Microprocessor and User Device Design Option 2 11](#_Toc521090746)

[6.4 Communication Between C++ and Java Applications 11](#_Toc521090747)

[6.4.1 Commnication Between C++ and Java Applications Functional Description 11](#_Toc521090748)

[6.4.2 Communication Between C++ and Java Applications Design Option 1 11](#_Toc521090749)

[6.4.3 Communication Between C++ and Java Applications Design Option 2 11](#_Toc521090750)

[6.5 Processing Algorithm 11](#_Toc521090751)

[6.5.1 Data Collection 11](#_Toc521090752)

[6.5.2 Programming Language 12](#_Toc521090753)

[6.5.3 Algorithm Implementation 12](#_Toc521090754)

[6.6 System Display 13](#_Toc521090755)

[6.6.1 System Display Functional Description 13](#_Toc521090756)

[6.6.2 System Display Design Option 1 13](#_Toc521090757)

[6.6.3 System Display Design Option 2 14](#_Toc521090758)

[6.6.4 System Display Design Option 3 14](#_Toc521090759)

[7. Team 14](#_Toc521090760)

[8. Final Prototype 15](#_Toc521090761)

[8.1 High Level Data Flow 15](#_Toc521090762)

[8.2 OBD II Scanner 15](#_Toc521090763)

[8.2.1 Hardware Design 15](#_Toc521090764)

[8.3 Data Acquisition 19](#_Toc521090765)

[8.3.1 Gathering Gyroscope data 19](#_Toc521090766)

[8.3.2 Gathering OBD data from the car 20](#_Toc521090767)

[8.4 Communication 20](#_Toc521090768)

[8.4.1 Sending data from the OBD II Scanner to the laptop 20](#_Toc521090769)

[8.4.2 Receiving the Bluetooth data in the C++ Console Application 20](#_Toc521090770)

[8.5 Java Application 21](#_Toc521090771)

[8.5.1 Decision Tree Implementation 21](#_Toc521090772)

[8.5.2 Data Processing 22](#_Toc521090773)

[8.5.3 Obtaining Current Gear Prediction 23](#_Toc521090774)

[8.5.4 Post-Processing for suggested action to driver 23](#_Toc521090775)

[8.6 System Display 24](#_Toc521090776)

[8.6.1 Java Swing GUI 24](#_Toc521090777)

[9. Testing, Data Analysis and Results 26](#_Toc521090778)

[9.1 Quantitative Testing 26](#_Toc521090779)

[9.2 Qualitative Testing 27](#_Toc521090780)

[10. Schedule 27](#_Toc521090781)

[10.1 Conclusions and Future Work 29](#_Toc521090782)

[References 30](#_Toc521090783)

# 

[The following sections must be included in your final report. You may add sections or subsections to suit the needs of your project, but do not omit any of the sections provided here, unless instructed to do so. Follow the instructions and examples in each section to build your report. If you insert material from other sources, make sure the format is consistent with this document’s original format. If you have trouble with the formatting, please consult your course instructor.]

# Executive Summary

Driving a manual transmission can be a very difficult task when first trying to learn the technique. It requires a good amount of vehicle knowledge and real-time car knowledge. The process of learning can be greatly assisted by a companion or mentor who is well versed in manual transmission driving. Although incredibly helpful, needing a mentor to help the new driver learn can quickly become a problem to those who do not have such a person. This is where our Manual Transmission Trainer (MTT) can fill the void.

MTT is designed to be an easily implemented system that will be compatible with almost all modern cars. MTT will provide the new driver with the insight needed to successfully drive the manual transmission vehicle. The MTT system involves several components: the OBD II Scanner, hardware processing module, and a system display to interface with the user.

The OBD II Scanner's main objective is to interface with the car. By doing so, the MTT will be able to access the data needed to instruct the user on how to drive the vehicle. The data interface between the OBD II Scanner and the car will be in real time and therefore can instruct the MTT system quickly enough to relay the instruction to the user. Additionally, this instruction needs to not distract the user from their driving responsibility.

In order to assess the MTT's utility, the output of the system was compared to manually inputted gear data and compared for accuracy. The results yielded an accuracy rate of 72%. After this test, some inaccuracies and optimizations were noted and then corrected for. After these improvements were implemented, the accuracy rate increased to 93%.

The overall results of our project successfully meet the expectations that our group had established at the outset of the semester. Given the brevity of the summer semester, achieving an accuracy rate of 93% along with some additional general functionality is a great outcome. With that said, there is still plenty of room for improvement. Minimizing the hardware and consolidating the executables would certainly increase the marketability of the MTT. Furthermore, successfully implementing the gyroscope along with decision tree optimizations could yield even more insightful information to display to the user.

# Problem Definition

Learning to drive a manual transmission car is a difficult process. If you do not have someone to be in the car with you while you are learning it becomes even more difficult. The driver has to learn how use all three pedals with both feet and operate the gear shifter all the while still paying attention to the road to ensure safe driving. Once a learner has some basic grasp of the technique, shifting strategies on real roads can also be puzzling to a new driver and some even counterintuitive. Making mistakes about shifting strategies can not only be damaging to your vehicle but also can put the driver in dangerous situations. All the difficulties associated with learning to drive a manual car demonstrate a clear need for a device that can alleviate some of the difficulties associated with operating a manual transmission.

It is assumed that the driver has basic knowledge on what the clutch does as well as the gear pattern of their vehicle. It is also assumed the driver knows how to execute a gear shift, however roughly. To best alleviate the pains of the learning process, this module has to be able to use real time data from the vehicle to instruct the driver. Using this real time data, the system will display to the driver a suggestion of no action or action, action being upshift, and the associated gear to shift too.

To meet these requirements, three main systems were designed. The first is hardware which includes a custom designed microcontroller, and purchased UART for the OBD II data, gyroscope/accelerometer/magnometer, and Bluetooth transceiver. All of these boards are incased in a custom designed housing. The second system is the communication software which is run on the microcontroller and on the display PC. The microcontroller will gather the data of interest from the car and then transmit the collected data over the Bluetooth transceiver. Once transmitted, a second set of software is utilized to receive the data from the Bluetooth. Once this information is received, it is relayed to the third system. The third system includes a decision tree to determine current gear and driving rules to display driver suggestions which will run on the display PC.

# Background

Though electric cars may be on the rise, the internal combustion engine is still the most common form of power generation for cars. Though in the U.S. most of these cars use an automatic transmission, there will always be people who want to drive a manual transmission. Currently about five percent of all the cars sold in the U.S. are a manual [1]. However, there are still a lot of manuals on the road from previous years and a lot of them are being driven by new drivers. The process through which a person has learned to drive a manual has not changed in years.

For most new drivers, someone who knows how to drive a manual is typically in the passenger seat helping to instruct. This can be problematic depending on the teaching ability of the instructor and their actual knowledge which can be incorrect. This can lead to new drivers driving incorrectly and damaging their engine, damaging their clutch/transmission, or putting themselves in dangerous situations when stalled.



Figure 1: Damaged Clutch

As you can see in Figure 1, the clutch is severely damaged from repeated stalls. Replacing a clutch is costly and time consuming. Minimizing the damage done to the vehicle while learning will give a longer lifespan to the clutch and transmission overall.

The literature out there on manual driving strategies is mainly for someone who knows how to drive and wants to optimize fuel efficiency, performance, etc. This literature was considered in our driving rules. The rules implemented in the MTT are a simplified version of the driving strategies presented in the literature.

We have found no device like what is described in this document. There are not even real-world applications of the best shifting algorithms we found from academia. The closest to what we propose that people offer is professional lessons on how to drive a manual transmission.

# System Requirements

## System

### The system shall include an OBD II Scanner

The user will be able to plug the OBD II Scanner into their vehicles OBD II port and connect it to the training module via DB9 cord. The module will be attached via Velcro to the door of the vehicle. The scanner will then transmit driving data real time to the training module via Bluetooth.

### The system shall include a wireless communication setup to receive real time data

The user will be able to receive the car's real time data that is collected from the OBD II Scanner. The system display is not included on the OBD II scanner so the data must be sent from the scanner to the system display. The system must transmit the data without wires to provide the user with convenience, and keep the user safe from wires which may obstruct their driving.

### The System shall implement a decision tree for determining current gear

The system will read in the data that is transmitted over Bluetooth to a Java application that implements a decision tree machine learning algorithm. Data that has been collected via driving around in every gear will be parsed into a .arff (Attribute-Relation File Format) file that is commonly used in machine learning. This file will be in the same directory as the Java app, and the trained decision tree will learn how to predict a gear based on this .arff file.

### The system shall include a PC display

The system will communicate with the user via a Java Swing (JSwing) application. The display will contain the current gear that the car is in according to the decision tree, the gear to shift up to whenever appropiate according to best practices of driving, a text box to record the actual current gear for testing purposes, and buttons to indicate the starting and stopping of driving.

## OBD II Scanner

### The OBD II Scanner must send data in real time

Real time data from the vehicle must be a part of the decision tree algorithm. Due to this, the OBD II Scanner must be able to send the data it is receiving from the car to the processing board with as little latency as possible. This data includes speed RPM's, throttle position, and torque.

### The OBD II scanner must rely only on power from OBD II port

The OBD II scanner will use the 12 VDC power available in the OBD II port.

### The system shall include an ODB II reading UART

The system will employ a pre-designed UART board to perform all the OBD II protocols and output the requested data to the microcontroller. The MCU will request a datum (RPM, speed, throttle position, torque) and the OBD II will output raw data to be arithmetically processed on the MCU.

### The system shall include Gyroscope/Accelerometer/Magnometer breakout

The box will have a gyroscope and accelerometer to collect data on the pitch of the road and if the car is speeding up or slowing down. This data will then be sent to the MCU for processing in the pattern recognition algorithm. The data of this pitch will be put through a low pass filter to prevent data shift over time.

### The system shall include Bluetooth breakout

The system will include a Bluetooth breakout that will output the data from the MCU to the PC display system.

### The system shall include a Custom Designed MCU

The system will have an MCU designed custom for the MTT. This MCU will be based

around an Arduino architecture with only the MTT required portions. This PCB will interact require 2 sets of serial connections pins and one set of I2C pins.

### The whole system shall be housed in a custom design box

All circuits will be housed in a custom designed housing

## Communication System

### The communication must be implemented using Bluetooth

The overall system must be wireless and thus Bluetooth was chosen as the communication modality. Bluetooth was the implementation chosen for several reasons. Bluetooth communication is a modern standard for many user applications. Thus, Bluetooth will be familiar to the user which translates to less of a learning curve.

### The communication must operate with a workable latency

The data from the OBD II Scanner is to be fed into the processing algorithm in real time. Due to this, the latency for the Bluetooth communication must be small enough that the data can be read, transmitted, received, processed, and displayed to the user quicker than common gear shifting times. The Bluetooth latency is of particular concern because it will most likely be the slowest portion of the whole operation.

### The communication must be reliable

The data from the OBD II Scanner will be accurate. Once transmitted, the data must remain accurate on the receiving end. Data accuracy is crucial to the system's reliability of instruction to the user. Ensuring that correct data is being received will ensure that the data being shown to the user is as accurate as possible.

## Data Processing and Algorithm Implementation

### The data from the OBD will be transmitted via Bluetooth and C++ program

The data that is read in from the OBD II Scanner will be in base 10 format and will be transmitted to the PC and Java application by a Bluetooth chip controlled by a C++ program embedded on the Boarduino. This program implements a pipe that are opened in the Java application and read in byte-wise.

### The data received by the Java Application will be parsed and processed

The data that the Java receives from the pipe will not be usable until it is parsed and processed by the Java application. The raw data that is received from the pipe will be in base 10 format and comma separated, so it will need split on commas into each string of data bytes that each contain a different type of value coming in (MPH, RPM, throttle position). There is an OBD PID standard for each type of value that is read from the OBD to generate the actual value of the type of data. These standards are universal and are found on the Wikipedia page for OBD-II PIDs [[2](https://en.wikipedia.org/wiki/OBD-II_PIDs)].

### The data that has been processed will be used to traverse the trained tree to find current gear

The data that has been parsed and processed by the Java application into their actual values will be used to create a new Matrix, which is the data structure used to compare the enumerations in the .arff file to the new data points coming in. The single line of data that comes in at a time will be combined with a static header identical to the one in the training .arff file. The Matrix will load the header and line, and then be passed to the method that traverses trained tree and returns the leaf node that it ended up at. The value in this leaf node that is returned is the predicted current gear according to the decision tree.

### The decision tree algorithm will be fast enough to not introduce additional latency

The data is read in at one line per second, so the decision tree must be able to be traversed and return a leaf before the next data point is read in from the pipe. In addition to reaching the leaf node before the next data point, this gear must be processed by the post-processing algorithm to return the predicted next or current gear to be displayed to the user.

## System Display

### The display will be the user's laptop

The display of the results of the system to the user will be on the user's laptop. The display will be created with JSwing, a simple GUI framework existing in Java. The display will appear when the Java application is run and the tree is trained.

### The display will indicate which gear the car is currently in

It can be challenging to be able to tell what gear you have the car in when you first start driving a shift until you get the feel for the car, so the display will also display what gear the car is currently in to the user so they can make a judgement on whether to upshift or downshift.

### The display will instruct the user on when to upshift and to which gear

When the user causing the RPMs of the vehicle to surpass the threshold for the current gear, the system display will have a notification letting the user know that it is time to upshift. This notification will be displayed above the current gear, will read "UPSHIFT TO: <next gear>", and will flash green in order to indicate an upshift is needed.

### The display will contain a textbox to input the actual current gear of the driver

The display will contain a small textbox that will be used for purely testing purposes and when a passenger is in the car. The point of this input is to record the actual gear that the driver is in to be compared to the gear that was displayed to the driver at the same iteration of the loop. This allows the developers to get an accuracy rating.

# Design Constraints

The following design constraints must be followed in order to produce a successful and sustainable product. These constraints focus on the safety and comfort of the user, the efficient use of energy, and the portability of the entire system.

## The system will not distract the user from their driving responsibilities

The system must convey information to driver without distracting their eyes from the road. The system's visual display must be noticeable from peripheral vision.

## The system will require no external power supply

The system will be fully functional running on power provided by the car and nothing else. The system will require no extra batteries or other charging devices.

## The system will be real time

The system will collect and process the data and then output driver action data in real time. The system will be fast enough to ensure the driver action information provided via the mobile app matches what the algorithm is saying the driver should do at that time.

## The system will be compatible with just one vehicle

For the scope of this project, the MTT will be designed around one vehicle. The ECE skills used do not depend on using many different cars. Complicating the system with adding in the mechanical parameters is well beyond the scope of a capstone ECE project.

# Evaluation of Design Concepts

## Overall System

MTT is composed of 3 main components: OBD II Scanner, wireless communications system, and PC display with decision tree. The high-level design for the system can be found in Figure 2 shown below. The user will be instructed on the best practices for shifting based upon the current situation. Data from the car's real time data as an input into a decision tree algorithm to discern the current gear which will be passed to a best shifting practices algorithm to determine what the user should do. The user will receive this output from the system via a PC display.

The real-time driving data will be taken from the OBD II scanner breakout using a serial connection. The accelerometer on the Gyroscope/Accelerometer/Magnometer breakout will be outputting real time data via I2C protocol on the current pitch of the vehicle on the road. Both the OBD II scanner and the accelerometer data will be sent to the MCU for arithmetic processing and finally to be sent via Bluetooth to the PC display program.

The machine learning algorithm will be one of the supervised variety. It will be written in a high level language like Java, and it will run on the user’s pc where the GUI will be generated. Data that has been collected from driving around with an OBD connected to a computer will be passed to the algorithm which will be able take in this data and learn how to predict the current gear of the car correctly. It will be able to make inferences and decisions based on the current conditions the car is experiencing and give a real-time output to the display to advise the driver on how to shift to the next gear.

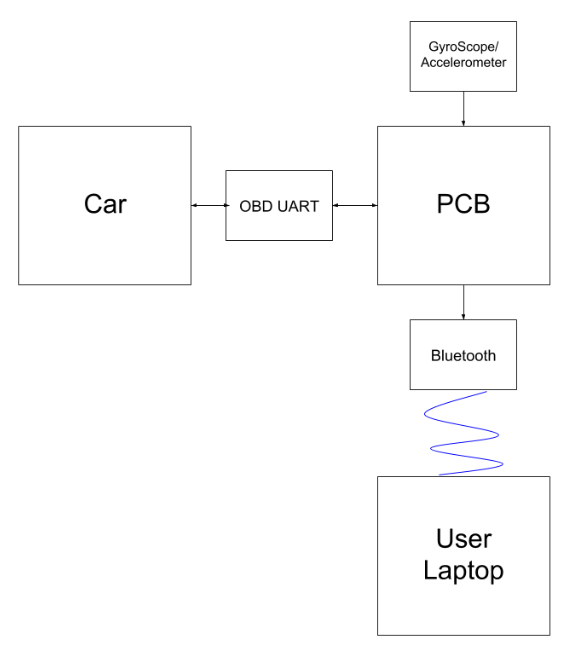


Figure 2: High Level Block Diagram

## OBD II Scanner

### OBD II Scanner Functional Description

The OBD II scanner component of the MTT is responsible for collecting the speed, RPM, throttle position, and engine load data from the vehicle itself in real time. Every vehicle manufactured after 1996 that was sold in the United States is required to have an OBD II port. This port is typically located underneath the steering wheel and can be utilized with a simple 16 pin connector. For consistency, every car was required to use a female connector. The OBD II port uses a J1850 protocol to send data to whatever device is connected to the port.

#### OBD II Scanner Design Option 1 - Off the Shelf Scanner

During the course of the designing the MTT, an off the shelf scanner was considered. This would mean using a device already completely designed to alleviate the team of any hardware design. These devices are usually inexpensive but are locked to use with limited software packages, so it would not be able to communicate with our own software.

An off the shelf scanner was decided against for not only the previous reasons but also because taking out all of the hardware design would eliminate an important part of an ECE capstone design course.

#### OBD II Scanner Design Option 2 – Completely Custom Design Scanner PCB

This option would have meant that the MTT would have a single PCB that would include all the required hardware for handling the J1850 protocols, running the gyroscope/accelerometer/magnometer hardware, as well as the MCU and Bluetooth hardware. During the design process, the team employed a SparkFun breakout OBD II scanner board and a SparkFun accelerometer breakout. While using and studying this hardware, it was determined that while the designing of putting all the hardware required for customizing both boards, assembling and soldering a board that would function as required would not be possible in the scope of the project. If the components size were increased, the MTT box would well exceed the size constraint on the box. For these reasons a completely custom designed PCB was rejected.

#### OBD II Scanner Design Option 3 – Custom MCU, SparkFun Breakouts

It was decided to use the SparkFun breakouts for the OBD II scanner, accelerometer, and Bluetooth and then custom design a MCU to operate the whole system. This design choice has many advantages for the MTT. First and foremost, using the OBD II scanner breakout will give us complete control over what data we will get from the car as well as allow us to send this data to our own program. The SparkFun breakout will fit within our size constraints for the vehicle. The accelerometer breakout offers the same advantage of having complete control on only using what we need while keeping size at a minimum. It is also advantageous to use these two breakouts because the OBD II uses serial connection and the accelerometer uses I2C connection which allows the two data collections to occur in parallel and to minimize the load on the MCU.

Custom designing the MCU based on an Arduino architecture to maximize efficiency and processing speed. The custom PCB was designed with only what is necessary for the MTT to operate and takes out all the components that make an Arduino versatile to other topics. Custom designing the MCU also will demonstrate hardware design skills as a portion of a capstone ECE project.

### OBD II Scanner Housing Functional Description

The physical box that will house the MTT hardware will have to comfortably fit inside a vehicle on the driver side of the car. The housing is self-contained with an open port to plug in the DB9 of the DB9-OBD II cord.

#### OBD II Scanner Housing Design Option 1– Machined Metal Box, Mounted on Port

Initially, the MTT was going to be housed in a machined aluminum box. This was eventually decided against because an aluminum housing would be less pliable to the MTT’s needs and would pose a risk of injuring the driver while it is resting on the driver side of the vehicle. This box was going to be hanging off the car’s OBD II port. This was not used in the final design because the size of the MTT was too big to safely hang off the port without interfering with driver pedal operation.

#### OBD II Scanner Housing Design Option 2 - 3-D Printed Box, Rest on the Floor

3-D printing the housing would offer the benefit of more control of the design to the MTT’s needs and would pose limited risk of scratching the driver’s leg while in operation. The design will be placed on the driver side floor and connected to the port via a DB9-OBD II cord. This design choice was made because the design challenge of getting an OBD connecter integrated into the design is outside the scope of an ECE project. It also still guarantees the accelerometer would be oriented in the same way during each driving session.

## Connectivity Between Microprocessor and User Device

### Connectivity Between Microprocessor and User Device Functional Description

The system as a whole will contain multiple independent sub-systems that will need to be interconnected. The connection between subsystems will need to have low latency communication in order to correctly instruct the driver. The two main connections being made are between the OBD II scanner and the Processing hardware as well as between the processing hardware and the system display. The latency that is of paramount importance is the above two connections together. Due to this, the design options are not mutually exclusive and the final prototype may end up using multiple options together.

### Connectivity Between Microprocessor and User Device Design Option 1

The system connections can be done via wires running directly between sub-systems. This option will certainly provide the smallest latency. Additionally, this option will be straight forward to implement. The data transfer between sub systems can be implemented simply with in and out busses. The major downside of this implementation is the presence of the wires. Wireless systems are more user-friendly implementations since they can be placed without having to worry about accounting for the wires.

### Connectivity Between Microprocessor and User Device Design Option 2

The system connections can be done via Bluetooth technology between subsystems. If the OBD II Scanner and the Processing Hardware box both contain a Bluetooth transceiver then none of the MTT's sub-systems need to be physically connected through wires. Bluetooth connectivity will provide the user with a simple and clean setup. The major downside to Bluetooth technology is that it cannot compare with the speed of hard-wired sub-systems. A major priority for the MTT is that it will relay the driving instructions to the user quickly enough so the user can act. Having multiple Bluetooth connections in series may not provide the necessary speed.

## Communication Between C++ and Java Applications

### Commnication Between C++ and Java Applications Functional Description

The data will be read into the user device with a C++ application and displayed to the user via a Java Application. These two applications will be running in separate processes on the CPU and therefore simple file input and output will not be sufficient.

### Communication Between C++ and Java Applications Design Option 1

The communication between the C++ and Java applications could be carried out via a Named Pipe. A named pipe is a form of interprocess communication (IPC) that can be used to send data between processes. For our implementation, the C++ would act as the client and would send its data to the Java application acting as the server.

### Communication Between C++ and Java Applications Design Option 2

The communication between the C++ and Java applications could also be carried out via the use of socket communication. Socket communication is where any process can send data over the computer's network socket ports. Sockets similar utility as the named pipes, but both have their own slight differences.

## Processing Algorithm

The processing algorithm will be the core of the project; it will need to be able to take in existing data of how a manual is driven properly and how automatic transmissions handle shifting gears, then use pattern matching to give the best possible suggestion of changing gears when considering a variety of variables.

### Data Collection

In order to accurately advise the driver on when to switch gears, the algorithm must have a good idea itself on how gears are switched normally. The more data that is available, the more accurate the algorithm will be.

#### Data Collection Method 1

Data that already exists from multiple resources (online, other students' data in SAE) will be manually uploaded to be processed by the same algorithm that would be used to calculate averages and identifying trends mentioned above.

#### Data Collection Method 2

Data will be collected with an OBD device and gyroscope while having a team member actually be driving the car. The device will record speed, RPMs, and throttle position while the gyroscope will record the grade of the car is driving on. This data will be parsed and fed to machine learning algorithm along with the actual current gear of the car in order to learn what the state of the car is matched to the current gear it is in.

### Programming Language

The code that will be the brain of the entire system will need to be either installed onto a microchip making this software embedded with no user interaction, or run on a PC in a high level language.

#### Programming Language Option 1

The first option for a language is simply C. C is the classic procedural language that is used most commonly in system programs still today. It is lightweight (needing little memory) in nature and can accomplish most tasks in a small amount of code compared to heavy languages like Java.

#### Programming Language Option 2

The second option for a language is C++. C++ offers essentially everything that C does only at a cost of being a little heavier than C. C++ is procedural like C, but it also offers classes and objects making it object-oriented as well. This allows for a little more versatility with less work than C.

#### Programming Language Option 3

The third option for a language is Java. Java is a high-level object-oriented language that contains many packages and prebuilt data structures to help the developer with accomplishing their task. The application written on Java would reside on the user’s PC, and would handle all of the processing of the raw data, as well as the decision tree and post-processing algorithm to determine best practices of driving.

### Algorithm Implementation

This algorithm will take care of all the real-time work that needs done while the driver is using the MTT. It needs to be thorough enough to make good suggestions on the gear switching, but also lightweight enough that it can take in parameters continuously and return values to the user in a time that will direct reflect the values that are being given to the algorithm at any given moment.

#### Algorithm Implementation Design 1

The second option is to use a pattern recognition algorithm that is unsupervised, such as a clutter algorithm. Unsupervised learning algorithms are not given any type of pre-labeled or preprocessed data to base new outputs off of; these algorithms are fed large amounts of raw data and try to make clusters based off of similarities in the data. In our instance, we would have to drive around a lot while collecting the gear, speed, acceleration, and grade of hill at any given moment. Then this data would be processed by the cluster algorithm and made into clusters similar to the classifications that we would define ourselves in a supervised algorithm. The advantage of this implementation is that there is a possibility of getting more accurate clusters from the algorithm than the classifications we create in the supervised algorithm. However, if we don't give it enough raw data – which will be time consuming to collect - then the clusters could be very inaccurate.

#### Algorithm Implementation Design 2

The first option is to use a pattern recognition algorithm that is supervised, such as a decision tree algorithm. Supervised learning algorithms are ones that use a known set of training data pairs that consist of an input and a desired output, and the algorithm produces an inferred data structure such as a tree that is used to map the incoming new data. In our instance, we would try to make the attributes of the training data based on the current gear, vehicle speed, engine speed, and throttle position of the car. The advantage of supervised learning is that it will produce an accurate output given that the attributes we chose to feed the algorithm vary in a direct way with the current gear of the car. The problem lies with the decision tree algorithm being able to learn how to predict the current gear accurately enough that the output will be logic data.

## System Display

### System Display Functional Description

The system will contain a method to display the output of the processing algorithm. The output being displayed will be in the driver's view and will instruct them on how to proceed. The display needs to be effective enough to correctly instruct the driver, but it cannot be too distracting as to take the user's attention off of their driving responsibility.

### System Display Design Option 1

The first option to display the information to the user is a mobile application. An example of what the mobile application would display is shown below in Figure 5. The display will show the current gear, a circle flash when it's time to upshift, and a graded portion which guides the user on how much throttle to provide. Smart phones and similar devices are pretty ubiquitous in a modern environment, and thus a mobile application would be an easy way to display the system information. Additionally, a mobile application display would be dynamic and laid out efficiently to instruct the user. One potential downside to this design option is that it forces the connectivity of the display to the hardware processing box to be done through Bluetooth. Also, it will take considerable time and man-hours in order to add a mobile application to the system.

### System Display Design Option 2

The second system display option would be an analog system consisting of LEDs to articulate the driving information. This design option would not require any additional hardware to use the system and thus could be used by those who do not have a smart phone. This universality carries some downsides as well. An analog display would require the hardware processing box to be hardwired to the display which is not ideal. Furthermore the analog display would require taking up additional and unnecessary space in the user's car.

### System Display Design Option 3

The third display option would be to use a GUI that is present on the user’s computer. Since a PC is so powerful, it can handle all of the data processing as well as the system display at the same time. This system display could be built with the Java Swing package which allows for simple UIs that allow user input and display output. This JSwing application can reside alongside of the machine learning algorithm which will allow for easy communication between the processed data and the display to the user

# Team

Ben is primarily involved with the software development side. As a computer engineer he has had some industry experience with ANSYS in developing their mechanical product. Additionally, he has done some back-end development with python for ANSYS. In addition to the software coursework and projects, Ben also has some experience in signals processing. Additionally, Ben is quite well versed in MATLAB which can be useful in various calculations.

For the MTT, Ben will firstly be responsible for writing the code to deal with Bluetooth connectivity. This will involve code snippets to both send the data from the Bluetooth transceiver to the laptop, as well as the code to receive the data on the laptop C++ console application. Ben's second responsibility is reading the data from the gyroscope and OBD port. This entails written the microprocessor code to communicate with the car are the gyroscope chip. Ben's third task will be getting the C++ application that reads in the data from the Bluetooth signal to communicate with the Java application that is responsible for the data processing and display. Additionally, Ben will assist in data processing in order to make sure that the final project can get finished in time.

Jimmy will also be focused on the software implementations of this project. He is a computer engineering major, so working with mostly code fits his skillset most logically. He is experienced in front end development, specifically in Java, JavaScript, and C#. These tools will be useful for building a computer application with good UI and logic.

Jimmy also has some experienced in using existing and creating his own algorithms to accomplish known tasks. This experience comes from previous classes and work experience at his co-op. Algorithm development will be needed for the MTT to accomplish things like taking in the specifications of the car for consideration on when to shift and taking in the grade of a slope that the car is on to more accurately advise on the shifting of gears.

Finally, Jimmy will be taking on the task of the development of the computer application that contains the machine learning algorithm and generation of the user interface. His experience in high-level languages will assist him in developing a comprehensive application that can handle all of the data processing of the data from the OBD scanner, as well as using this processed data to display suggested actions to the user in understandable and logical manner.

Nick will handle all of the hardware design. This includes the electronics inside and the physical design of the module. He will be responsible for ensuring the MTT can run with in the given power constraints and that it will not overheat. He will do the schematic and design of the PCB in the MTT. Nick will also use his experience using 3-D modeling to design the housing of the MTT and the mounting mechanism as well as ensure proper cooling is taking place. Nick will be able to develop the algorithm for shifting strategies on a high level as he has been driving manuals for over three years. Nick also has a general knowledge of cars and how they work. Nick will also be involved in making the sure the data is displayed in a way that gets the information across without overly distracting the driver from the road.

# Final Prototype

## High Level Data Flow

Refer to Figure 2 for the High Level block diagram of the system.

The accelerometer component of the MPU9250 breakout is used to determine the pitch of the vehicle on any given road. The pitch is the angle from flat the road makes and can be positive or negative. Pitch will be examined in greater detail later in this section. The data collected from the MPU9250 accelerometer has to put through a low pass filter to prevent data drift.

The final decision on the MTT’s output is decided from the data on given by the decision tree and then evaluated by the ideal driving rules. These rules were determined from academic literature [3] In later parts of this section, these rules are explained in greater detail.

## OBD II Scanner

### Hardware Design

The MTT design starts with the hardware in the actual module. This includes the SparkFun OBD II UART breakout, MPU9250 breakout, Bluetooth breakout, and custom designed MCU. These devices handle all data collection, arithmetic pre-processing, and wireless data transmission to the PC. These devices are all housed in a custom designed and printed plastic box.

#### Hardware Schematic

The schematic below shows the overall schematic for the MTT’s hardware. All portions are considered “black boxes” in this schematic to show how the system as a whole is wired. This was designed using EagleCad Software

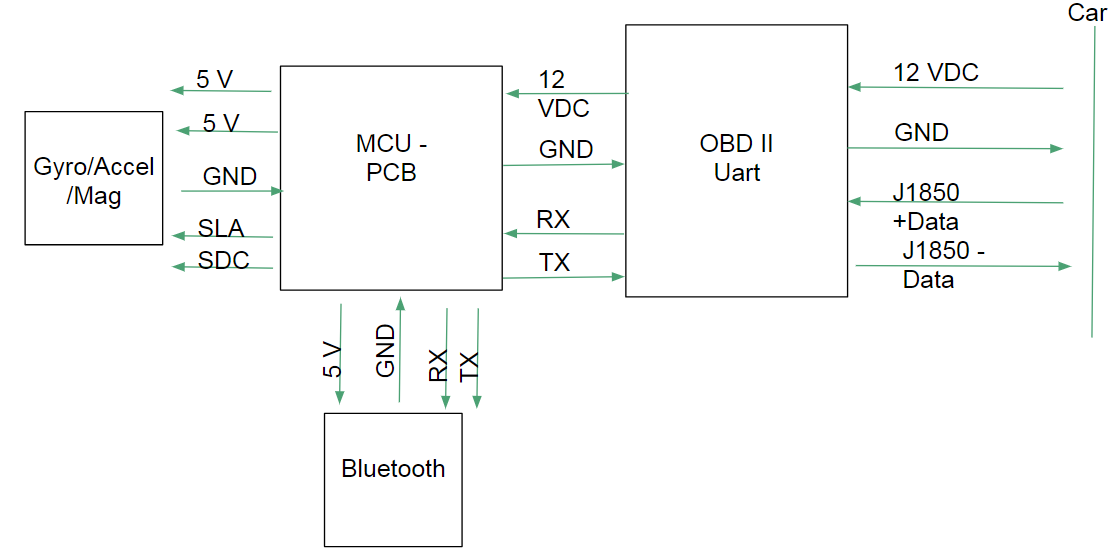


Figure 3: Hard Schematic

##### Custom PCB

The circuit is primarily based on a Adafruit Boarduino. The system operates as normal using two serial ports for the bluetooth and UART and one I2C port for the MPU9250.

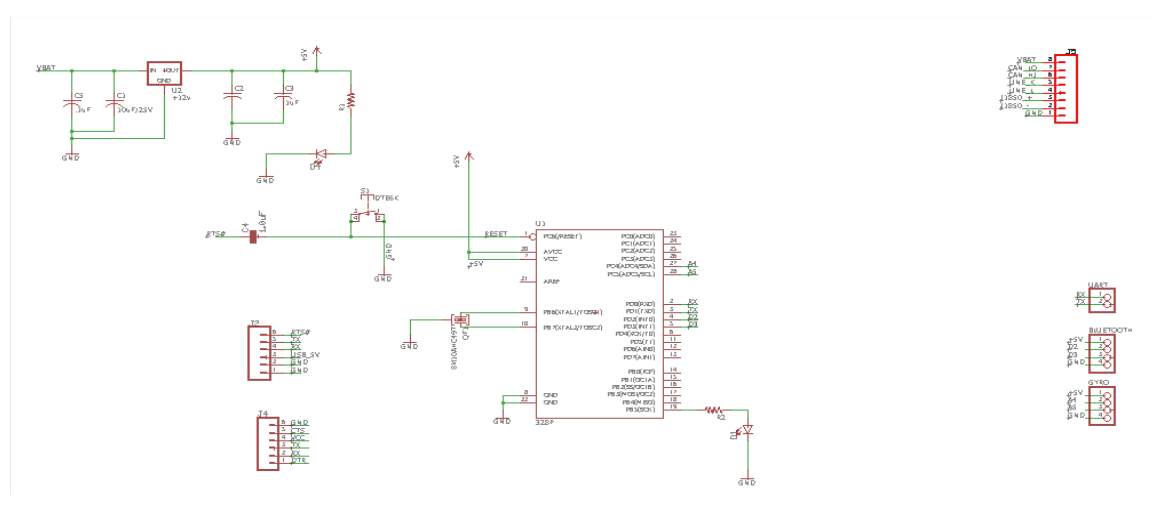


Figure 4: PCB Schematic

##### Power

All the circuits are powered from the 12VDC output in the vehicle’s OBD II port. The devices in the MTT run on 5VDC or 3.3VDC so all employ a voltage regulator from the power source, VBAT. The SparkFun breakouts all handle their own voltage regulation and the custom MCU has a 12V/5V voltage regulator built in. VBATT goes into the OBD II UART and is pulled from there to the MCU where it is then distributed out to the MPU9250 and Bluetooth.

#### MTT Housing

Below are captures of the main box and lid components of the housing for the electronics. Underneath are captures showing the dimensions of the components of the housing. Both were designed in SolidWorks and 3-D printed in the SERC.

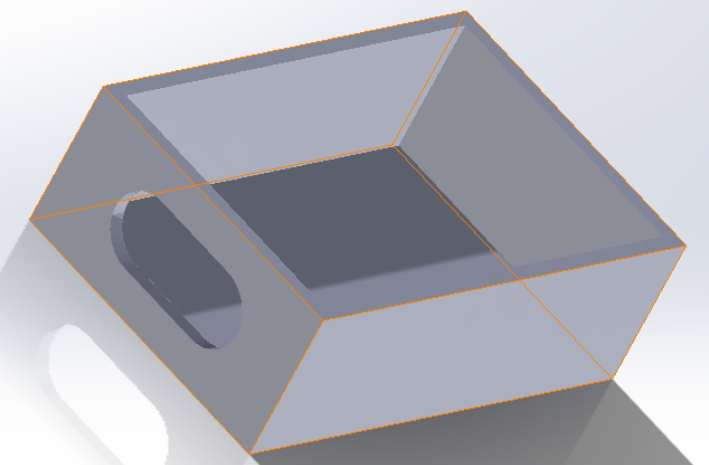


Figure 5: SolidWorks Design of Housing Base

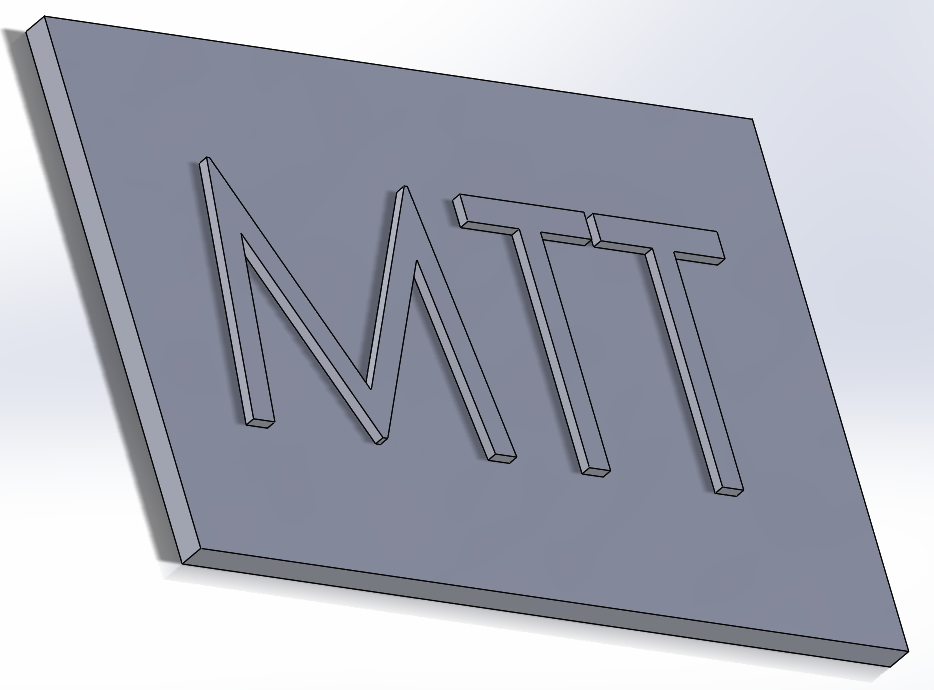


Figure 6: SolidWorks Design of Housing Lid

The electronic boards were mounted in the box using .5 cm offsets. Pictures with the lid on and off of the final prototype are shown below.

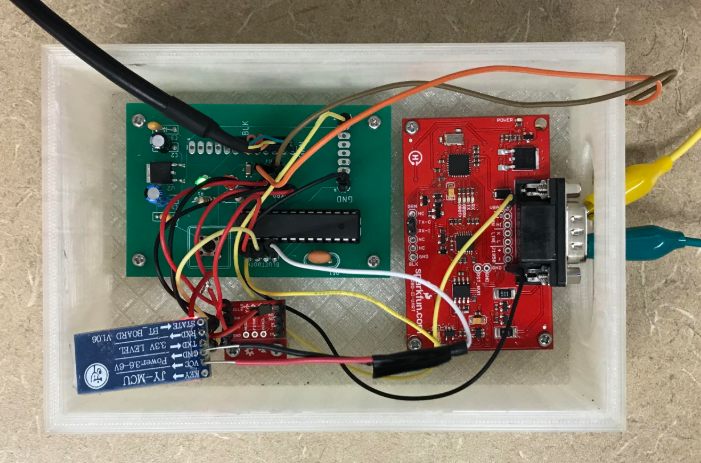


Figure 7: Actual Insides of MTT Module

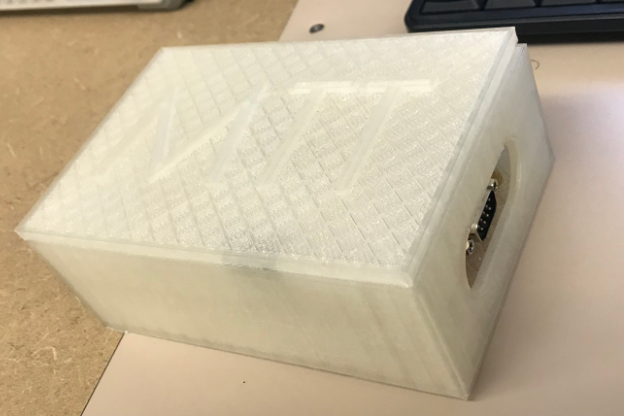
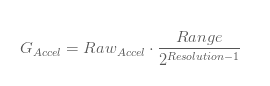


Figure 8: MTT Housing With Lid On

#### MPU9250 Accelerometer Filter

The data the MPU9250 cannot be directly used because way the accelerometer chip functions. The pitch is calculated using the differing force of gravity in the three directions. As the pitch changes the data will start to shift and give incorrect readings.

As described above, the calculation for pitch is done by measuring the different force of gravity in all three directions. The formulas used to calculate the pitch are from [4] and shown below:





To avoid the data shift described above, a simple low pass filter was implemented. This prevents the shift in data by ensuring that as time goes on changes are limited in scope, so the data doesn’t change by more and more magnitude. A diagram for the filter is shown below [5]:

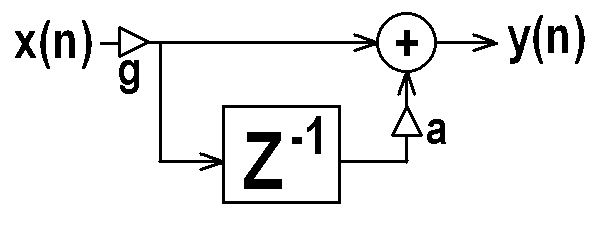


Figure 9: Diagram of Low Pass Filter

## Data Acquisition

### Gathering Gyroscope data

The gyroscope data was relatively straight forward to gather. Firstly, some sample header files were required to ensure proper operation of the chip [6]. The first step was to initiate some loop variable and initiate an MPU9250 class object. From there, the chip needed to be calibrated with sensor resolutions, bias, and scale. This is crucial to ensure that the data we receive from the chip during operation is both accurate and consistent. Once everything is initialized, the values are simply read from the chip if they are they are in the object's buffer. Once read, the final step is to calculate pitch. Pitch is the amount by which a vehicle is angled about the transverse axis as shown below in Figure 10. Unfortunately, the gyroscope had some unexpected errors in the days before final testing and had to be omitted from the final data collection.

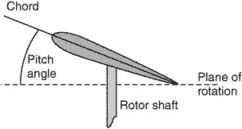


Figure 10: Visual Representation of vehicle pitch

### Gathering OBD data from the car

The car's OBD data was gathered through a specialized OBD Universal asynchronous receiver-transmitter (UART) that has been built to connect to the car's OBD II port. The UART was connected directly to the board's RX and TX pins to be used for serial communication. Skeleton code was used [7] to help assist in getting the correct reading procedures and results within a short timeline. The general flow of information was to send the UART a command to which it would echo back the command, print the data of interest, and then print a black line. For each of these responses from the UART, the function "getResponse" is called to read the serial port until a carriage return appears. The first step was to send the command "ATZ" in order to reset the UART. From there, the UART was used to collect the vehicle speed, RPM, throttle position, and engine load. These values were read in as singular (speed and throttle) or multiple (RPM and engine load) hexadecimal bytes that were parsed into integer values. These integer values needed further processing, but this was decided to be done within the Java application. At the end of every iteration, these values were added to the CSV string along with the gyroscope values.

## Communication

### Sending data from the OBD II Scanner to the laptop

The connection from the microprocessor to the Bluetooth transceiver was completed using a package called "SoftwareSerial". This package allowed us to create a serial communication to the Bluetooth based on pins that were not inherently to be used as RX and TX. This was a necessary step, as the built-in serial pins were already in use for the UART communication with the car. Once the data has been successfully queried from the UART and gyroscope, it was placed into loop variables as hexadecimal values to be stored for the current iteration. At the end of each iteration, the data is printed to the Bluetooth's serial bus as a comma separated line with the characters ">/n" at the end of each line. The commas are used to deliminate between data points, and the ">\n" is used to deliminate between each line.

### Receiving the Bluetooth data in the C++ Console Application

The data from the Bluetooth transceiver is sent wirelessly to the laptop's Bluetooth chip and is available to be read over a COM (Communication port) port. In order to read this data, skeleton code [8] for reading COM ports was sourced to assist in ease and speed of production. Additionally, the data also needs to be sent from the C++ application to a Java Application which will process and display the interpreted results. In order to complete this step, a software pipe was used. A software pipe is a means of inter-process communication (IPC) that can be used to send or receive data.

The first task is to initiate the SerialPort class to connect to the Bluetooth's COM port and initialize the software pipe to communicate to the Java processing code. Once initialized, the code will loop indefinitely so long as the microprocessor is connected and the pipe remains valid. Next, the data that had been written to the buffer between loops gets read into a buffer. The buffer is then parsed into the individual lines of data. From there, the second line of data gets written to the pipe which is connected to the Java processing code. The second line was chosen because only one line of data needs to be sent per iteration, and the first line of data is often partial because of the serial nature of COM ports. Once the data is written to the pipe, the loop variables are reset and the programs sleeps before entering into another iteration. 1000ms was used because it enabled the data to be sent as fast as possible without causing partial data to be sent.

## Java Application

### Decision Tree Implementation

After much consideration of multiple classification machine learning algorithms, it was decided to implement a decision tree as it seemed to fit our needs the best. The decision tree is a widely used machine learning algorithm that has existing implementations that are flexible in what they are able to take in as data to use for training and traversing the tree. The implementation used was an existing one that was found on GitHub (<https://github.com/Steve525/decision-tree>).

This algorithm uses the ID3 version of a decision tree to build and make decisions with. The pseudo-code for this algorithm can be found on the ID3 algorithm Wikipedia page and it is quite easy to understand at a glance [9]. The entire project that the decision tree is implemented in has a lot of fluff that isn’t necessary for the tree to work, but proved to still be quite useful for our use of the tree. The main file that it is run from, MLSystemManager.java, contains an argument parser and logic to train the tree and test data. An object of the MLSystemManager was created in the Manual Transmission Trainer’s main method, and it is used to help with the creating of the trees and sub trees of the real-time data. The Node class, which is the object class that the actual tree is stored in, has a method that creates a .dot file (graphic descriptive language) while the tree is being trained. This allowed us to generate a visual image of the trained tree to see how the exactly the tree was traversed to land at a certain leaf node. A very small portion of the tree (about 1/200th) has been include in Figure 11.

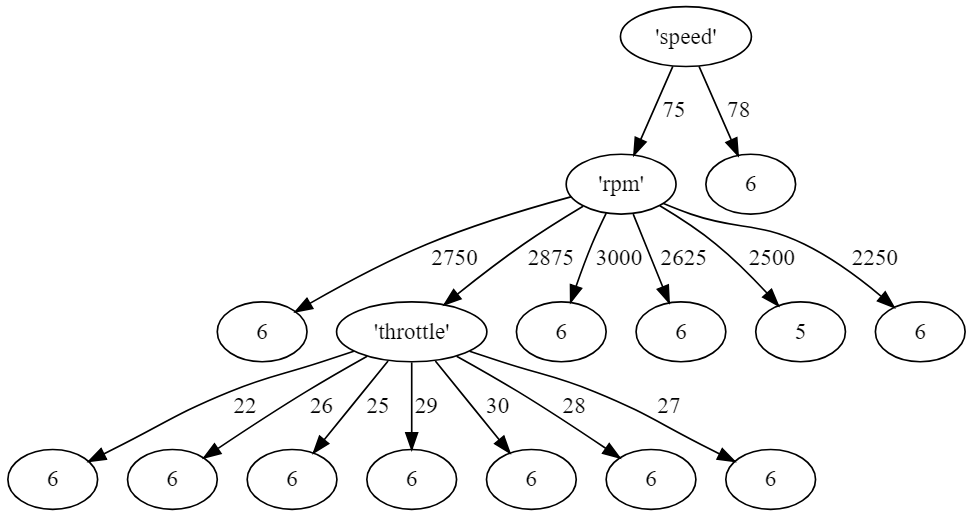


Figure 11: Small section of trained tree

The other files in the Main and decisiontree subdirectories of the source code are internal classes that are used to help construct the tree and make decisions with the tree. Most of the files in these two subdirectories had to either be altered or added to in order to be able to use our data in the best way possible. For example, in the Matrix class, methods had to be added to parse a header of the arff file separately from the entire arff file and the data itself. Then another method had to be added in order to parse the actual line of data into the Matrix. These methods are original, but they are based on the logic of how the Matrix parsed in arff files originally.

The GearFrame and TransmissionTrainer classes are completely original classes that were made from scratch. The GearFrame class is what builds the GUI of the application, and it is instantiated in the TransmissionTrainer. The TransmissionTrainer class holds all of the logic to run the application from start to finish. If the last argument the program is started with is “gearHeader.arff” the application knows that this means real-time data will be read into the system and this is what is run to generate the GUI and have the user use while driving. It will open up the pipe and process that incoming data that is explained in greater detail below. Otherwise, the application assumes it is being used for testing purposes and the last argument will be a trace arff file that will simply print out predicted gear values to be compared with known gear values in Microsoft Excel.

### Data Processing

As mentioned above in the Communications section, the data was sent from the OBD and MCU to the C++ application on the computer via Bluetooth in base 10 format, which was then sent to the Java application via an opened pipe. Although the data is integers at this point, it is not the integers that tree is looking for. These raw lines of data that are coming in one time a second are passed to a method in the Java application that use the *Integer.parseInt(<base 10 integer>, 10)* method to parse the already existing integers into different integers that are then entered into an equation to determine the actual value of the attribute that is being processed. For example, since RPMs are three or four digits, the RPM is data is sent in two groups of two integers, which are individually parsed, and then given to the equation ((RPM1 \* 256) + RPM2)/4. This will produce the proper three or four digit RPM integer that is to be given to the decision tree.

### Obtaining Current Gear Prediction

After the data is processed from the raw string of data to the proper integers, this string of data is combined with the proper .arff file header each iteration in order to produce a Matrix object of the data. The Node object that the trained tree resides in uses Matrix objects to align the enumerations that are present in the .arff header with the internal and leaf nodes of the tree. The Node class contains a method called *makeDecision* that takes the first row of the Matrix which are the features, or attributes, of the data as well as the trained tree’s attribute’s position in the column in order to compare all of the data in that certain attributes part of the tree with the incoming real-time data. This method returns a double value that is the contained in the leaf node that the traversal of the tree ended up at. This value is the prediction for what the current gear.

### Post-Processing for suggested action to driver

This prediction for the current gear is then used alongside of the car’s RPMs (and gyroscopic data, had it worked at the end) in order to determine what the driver is currently doing and what they should do next. This post-processing boiled down to a simple if-statement that can be seen in Figure 12. This if-statement simply checks to see if the car’s current RPM is in certain ranges, then tells the user what to do based off of that. If it is between 1150 and 2875 RPMs, then the UI displays the driver’s current gear according to the tree, and does not tell the driver to do anything. When the RPMs go to 3000 or higher, the current gear is still displayed, but an additional message appears in green and tells the user to “UPSHIFT TO: <current gear + 1>.” At this time, the driver should press the clutch and begin changing gears. Once the RPMs drop in the next gear, the gear that the driver was told to switch to should be displayed in the current gear text box with no suggested user action. If the RPMs ever drop below 1000 RPMs, it is safe to assume that the driver is in neutral and a message will display “Car is in neutral” and the current gear text box will display 0.

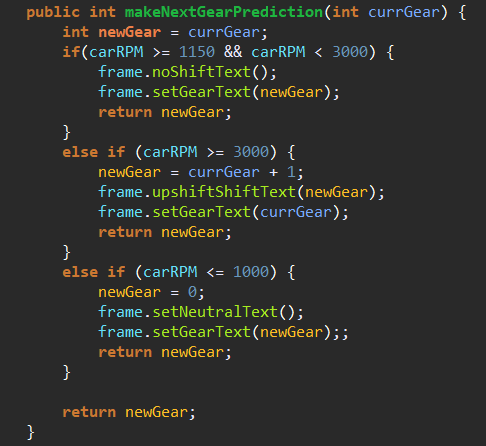


Figure 12: Post-Processing If-Statement

## System Display

### Java Swing GUI

Initially, we wanted to design a mobile phone application to display what is needed to the user while they’re driving since it is easily portable and easy to see while driving. However, due to lack of time and human resources, we were unable to get to the mobile application. Granted another member of our team, we believe a mobile application would have been a very viable possibility. Instead, we chose to design a Java Swing user interface that resides directly in the Java application (GearFrame.java).

The display is designed to have three different states essentially. First, when the car is assumed to be in neutral, a textbox at the top will read “Car in Neutral” and have a “0” in the current gear textbox. This can be seen in Figure 13. When the user is driving, if they are in the range of RPMs that determines no shift in gear is needed, it will display “Current Gear:” at the top and the current gear of the car according to the decision tree in the current gear text box. This can be seen in Figure 14. Lastly, when the car’s RPMs hit the range that a gear shift is advised, the text box at the top will turn green and display “UPSHIFT TO: <current gear + 1>.” The current gear will remain in the current gear text box and should change to the new gear once the RPMs drop from the shift in gears. This can be seen in Figure 15.

Finally, there is a small textbox at the bottom of the GUI that allows the developers to enter the actual gear of the driver while using the program for testing purposes. While running, the application prints to a new text file comma separated values of the predicted gear at the time and the actual gear of the driver that a passenger has entered into the textbox. This allows for an easy import to Excel in order to conduct calculations for test results.



Figure 13: Car in Neutral Display

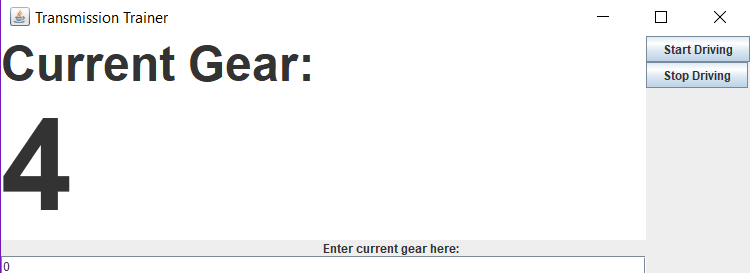


Figure 14: Current Gear Display

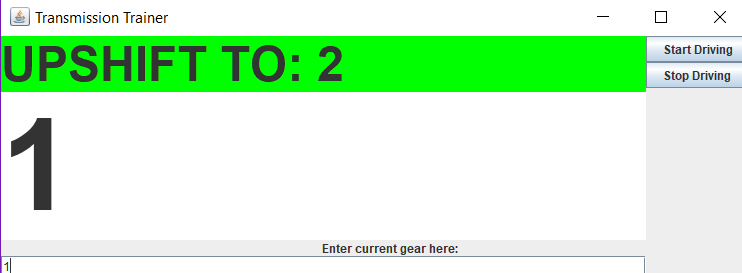
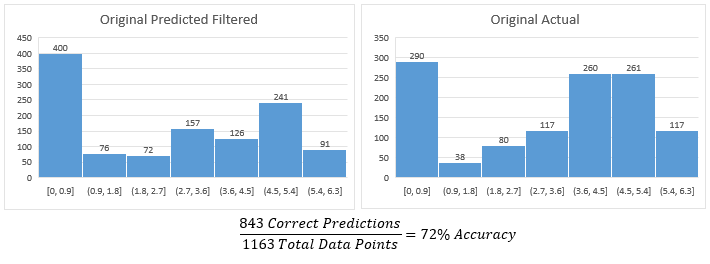


Figure 15: Upshift Display

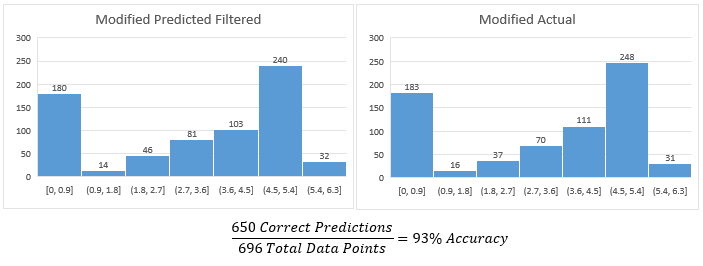
# Testing, Data Analysis and Results

## Quantitative Testing

Testing the MTT was tested by adding an input box to the JSwing display which would take in the manual input of which gear the driver was currently in. This value is stored until the driver switches gears and the new gear is then manually inputted. For every iteration of the loop, the predicted gear from the tree and the manually inputted gear were written to a CSV file. These values were then opened in Microsoft Excel to run statistical analysis. One important note is that we filtered the predicted gears so that if the vehicles RPM was less than or equal to 1000, we overrode the tree prediction and manually predicted the gear to be 0, or neutral. This was implemented based on the observation that the car is most definitely in neutral when the RPMs are this low, but the tree does not always pick up on this. The results of our initial testing are shown below in Figure 16.

Figure 16: Data for initial testing

Our initial testing yielded an accuracy of 72%. Upon further inspection, we noticed several things that were negatively impacting our results. Firstly, the number of predicted gear 0 readings is much greater than the actual number of 0 data points. We attribute this to two factors. Firstly, is that the engine load value had been read in differently during testing than the values that were used to train the data. We are unsure why this happened but the engine load value does not play a crucial role in predicting the correct gear. Secondly, we noticed that we were using speed as our first decision tree branch, when we feel as though RPM would make a better initial branch. These realizations were made whilst testing and, due to time constraints, we were unable to reimplement how our data was collected and used within the decision tree. With that said, we were able to change the JSwing program to output the raw data with the predicted and actual gears for the last serval test logs. This update allowed us to go back and modify the training data and the decision tree so that we could see what would have been displayed if we had gotten a chance to update all the source codes. These results are shown below in Figure 17.

Figure 17: Data for updated testing

Our updated testing results showed a much better overall result of 93% accuracy. These results imply that our changes of moving RPM to the first tree branch and removing engine load from the decision tree was a substantial improvement.

## Qualitative Testing

In addition to the quantitative analysis of how accurate the decision tree was in real time, we also did some qualitative analysis of how reliable the conditions, as shown in Figure 11, for shifting were. We had the display tell the user to upshift when the RPM's were over 3000. This condition seemed to be correctly carried out when the driver was indeed approaching an RPM value of 3000. Additionally, we had filtered the tree output so that when the RPMs were less than or equal to 1000, the car would be assumed to be in neutral. This also appeared to function well, as there were no instances of higher gears being displayed when the RPMs were indeed below 1000.

# Schedule

**Week of May 28th**

* Finalize Conceptual Design
* First part order for data collection
* Start hardware design
* Research classification-based pattern recognition algorithms
  + Decide on best approach

**Week of June 4th**

* Work on schematic for PCB
* Start coding Bluetooth communication
* Begin coding of algorithm

**Week of June 11th**

* Work on algorithm
* Continue refining Bluetooth communication
* Continue PCB design

**Week of June 18th**

* Finish algorithm
* Begin work on main Java application
* Finalize Bluetooth communication

**Week of June 25th**

* Using Bluetooth to write sample data to a file
* Begin Arduino code to use Gyroscope/Accelerometer
* Work on main Java application

**Week of July 2nd**

* Refine Arduino code to correctly calculate pitch
* Begin Arduino code to read OBD data via the UART
* Collect training data for decision tree
* Continue work on main Java application

**Week of July 9th**

* Further Refine Bluetooth communication
* Finish Arduino Code to calculate pitch from gyroscope/accelerometer
* Continue working on code to read OBD data from UART
* Design user interface

**Week of July 16th**

* Successful communication between C++ and Java applications
* Finish code to read OBD data from UART
* Use recorded data to identify classifications for algorithm
* Order PCB and parts for module
* Java application is able to read in data from C++ pipes and make predictions

**Week of July 23rd**

* Debug and refine entire system
* Java application altered to produce output files used for testing and ease of getting test results
* Complete project
* Design and print housing

**Week of July 30th**

* Last second revisions to project
* Report, Presentation, and Poster Creation

## Conclusions and Future Work

Altogether, the MTT came together with a sufficient accuracy of slightly over 90%. Given the limited timeframe, these results are very satisfactory. For future work, several improvements can be made. One of the first improvements to be made would be to correctly implement the gyroscope/accelerometer. Correcting this issue should be relatively straight forward with proper time to debug and assess what was causing the issue. The next thing to be done would be to consolidate and minimize the hardware. This step is crucial towards making the product more appealing to the everyday consumer. Another step towards simplifying user interaction would be to consolidate both the C++ and Java applications into one executable. The ability to execute only one file is paramount towards usability, as the current initialization is not suitable for user interaction. One final addition towards operation would be to include down shifting functionality. Instructing the user as to when to downshift is impractical because that would require knowing when the user has to slow down. With that said, functionality to instruct the user on which gear to switch to after performing a downshift is very doable and would be in the scope of the MTT.

# References

[1] Evarts, E. C. (n.d.). Why Are Manual Transmissions Disappearing? Retrieved May 27, 2018, from <https://cars.usnews.com/cars-trucks/best-cars-blog/2016/09/why-are-manual-transmissions-disappearing>

[2] <https://en.wikipedia.org/wiki/OBD-II_PIDs>

[3] Ngo, D. V. (2012). Gear shift strategies for automotive transmissions Eindhoven: Technische Universiteit Eindhoven DOI: 10.6100/IR735458

[4] <https://www.nxp.com/files-static/sensors/doc/app_note/AN3461.pdf>

[5] Corinne, Toni. “SparkFun OBD-II UART.” Alternating Current (AC) vs. Direct Current (DC), Sparkfun, learn.sparkfun.com/tutorials/obd-ii-uart-hookup-guide.

[6] .Brent. “SparkFun IMU Breakout - MPU-9250.” SparkFun, learn.sparkfun.com/tutorials/mpu-9250-hookup-guide\_ga=2.54184680.150780981.1533219235-36315267.1533070464.

[7] Dubois, Chantelle. “Programming Languages for Embedded Systems 101: Background and Resources.” All About Circuits, 26 May 2017, [www.allaboutcircuits.com/news/programming-languages-for-embedded-systems-101-background-and-resources/](http://www.allaboutcircuits.com/news/programming-languages-for-embedded-systems-101-background-and-resources/).

[8] Mandal, Manash Kumar. “Serial Communication with an Arduino Using C++ on Windows.” Manash's Blog, Medium, 22 May 2016, blog.manash.me/serial-communication-with-an-arduino-using-c-on-windows-d08710186498.

[9] <https://en.wikipedia.org/wiki/ID3_algorithm>