BinBot
Test Report

REVISION HISTORY

Revision #	Author	Revision Date	Comments
0.1	Jose Silva	December 8, 2019	Initial template
1.0	Jose Silva	December 8, 2019	Arm / Distance Sensor Test
1.1	Michael Savitski	December 8, 2019	Machine Learning Test
1.2	Kwamina Thompson	December 8, 2019	Android Mobile Application
1.3	Sean DiGirolamo	December 8, 2019	Server, Integration & Acceptance sections
1.4	Sean Reddington	December 8, 2019	Tread and Camera interfaces, formatting

Table of Contents

System Overview

BinBot is a waste-collection robot intended to patrol specific areas such as university grounds, stadiums, boardwalks, or schools. BinBot will identify waste that is laying on the ground and collect it to be disposed of properly. BinBot will have a camera module and on-board microcontroller unit that communicates with a server via wi-fi. To outsource the heavy data processing from the on-site robot; a Linux server will process the images sent by BinBot using data representations created with a deep learning algorithm to identify pieces of waste. The server will then inform BinBot of information regarding the photos, such as if any waste has been identified, and if so, how far it is and how BinBot should navigate to the waste. The waste will then be collected using a mechanical arm. BinBot will be able to be powered on/off via a mobile application.

One of the key components for BinBot to function as needed will be deep learning software for training a neural network model, so that BinBot's other software will be able to identify waste objects in the images from its camera module. This will entail the use of the OpenCV library for processing images, and the use of the open-source deep learning library TensorFlow. Ideally, this separate software component will be run on a GPU supported computer system. The neural network data model will be trained using hundreds to thousands of images of waste objects, captured in the expected resolution of BinBot's camera. Also included in the training data with these images will be information such as camera height, object size, and object distance, so that BinBot will be able to make estimations to allow it to traverse to the waste objects and successfully collect them. After feeding images to the neural network, the neural network will be tasked with returning whether or not waste is located in the image, and if so, how far the waste is and at what angle it is from the robot.

The completed data model will be applied to software which will run on a server along with a communication socket service, which BinBot will continuously connect to via wi-fi. The images continuously collected by BinBot's camera module will be sent to the server for processing via the OpenCV library, which will use the trained neural network data model to identify waste objects. Once the waste has been identified, as well as the distance and angle, movement instructions will then be calculated and sent back to BinBot's on-board computer board so that it may travel to the nearby waste objects and pick them up. For the purposes of this simple project, BinBot will simply be instructed to turn towards the waste, and travel to it in a straight line.

BinBot's on-board computer will be a Raspberry Pi 3 B+ board with limited functionality. It will be to collect images from the camera module and then transmit them to the data processing API. Receiving data back from the data processing API about waste objects in BinBot's camera's view, the primary purpose of the board will be to operate BinBot's robotic components. BinBot will have robotic treads for traversing across the floor to approach waste objects, as well as a robotic arm for collecting the objects. The software running on the board

will need to use the constantly updated information about distance and size of identified objects to operate these components efficiently and successfully collect waste.

Additionally, a companion mobile application will be developed for users of BinBot to install on Android devices. This companion application will be able to power on/off the server. The server will modify these images visually to have boxes around identified waste objects, and text about BinBot's current progress in its process of collecting the objects.

System Block Diagram

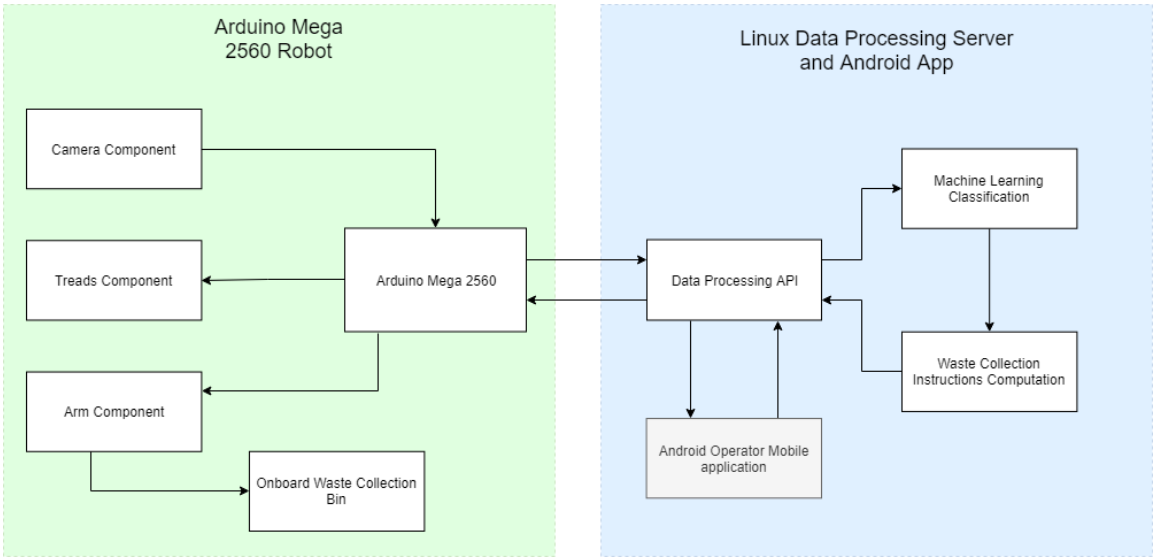

Fig 1. Block diagram illustrates the flow of data from the Camera feed to BinBot disposing of the waste.

The block diagram shows a high level overview of BinBot project's system architecture. Each of the hardware components on the robot kit will have a corresponding controller interface in the Sketch program loaded onto the Raspberry Pi. The Raspberry Pi will exchange data with the Linux server over Wi-Fi via a data transmission interface. The Linux server will be hosting a data processing API that will pass BinBot's camera feed to the machine learning classification processing. If there is collectable waste, the server will then computer instructions for the robot to collect the target waste object and send it back to the Raspberry Pi. The Raspberry Pi will then pass the instructions to the tread and arm interfaces to collect the object.

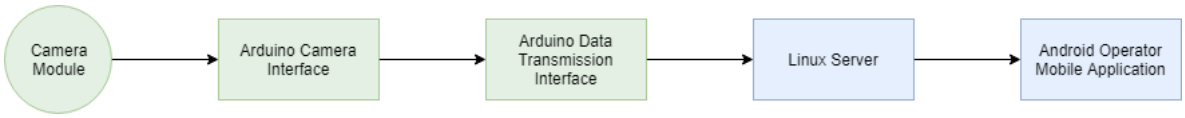

Fig 2. Diagram displaying interaction between BinBot's Camera Module, Linux Server, and Android Application.

The camera module will capture images at 1 fps via the Camera Interface. This image feed will be passed to the Raspberry Pi's Data Transmission interface which will send the images to the Linux server to be processed. Along with the data processing, the image feed will be passed to the Android operator mobile application for monitoring and manual intervention of BinBot during testing.

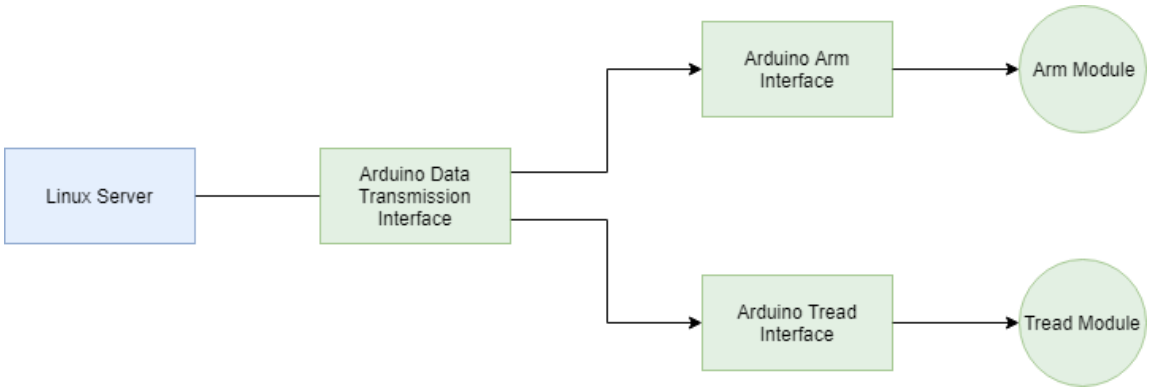

Fig 3. Diagram displaying interaction between the Linux Server and the Raspberry Pi's interfaces in order to operate the robot.

The Linux Server will exchange data with the Raspberry Pi over Wi-Fi through the Raspberry Pi's Data Transmission interface. This includes computed instructions for BinBot to travel to or collect the target object. The Raspberry Pi will then pass the instructions to the corresponding interface. These interfaces will provide the functions to move the arm or tread modules.

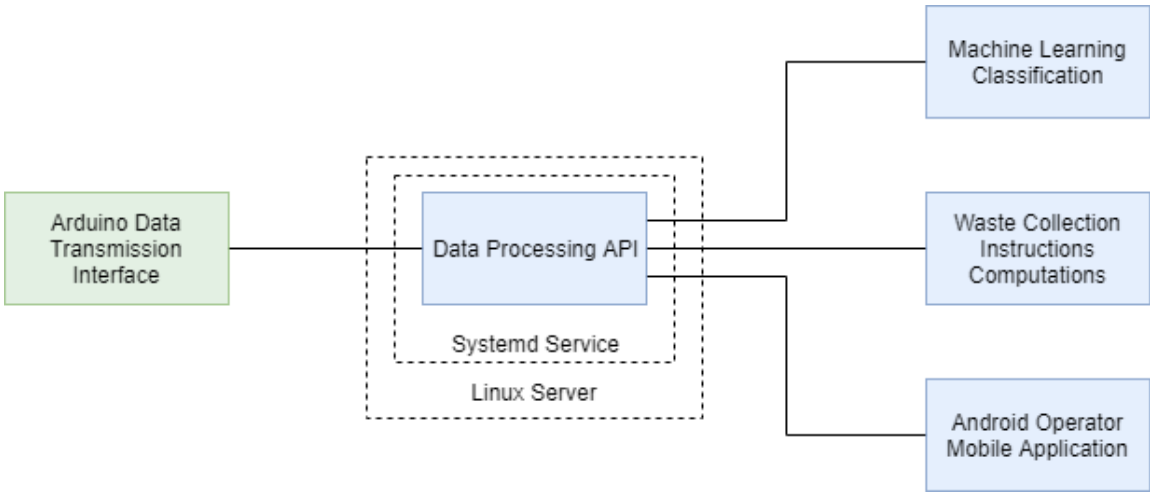

Fig 4. Diagram displaying the interaction between the Arduino's Data Transmission Interface, Linux data processing server, and Android Application.

The Data Transmission interface will send the image feed to the Linux processing server. The proposed method is to host a data processing API application that will execute via System services. The data processing API will relay the image feed to the machine learning classification processing. If the machine learning model identifies collectable waste, instructions for the robot kit to collect the waste will be computed. These instructions include navigation instructions for the tread module and collection instructions for the arm module.

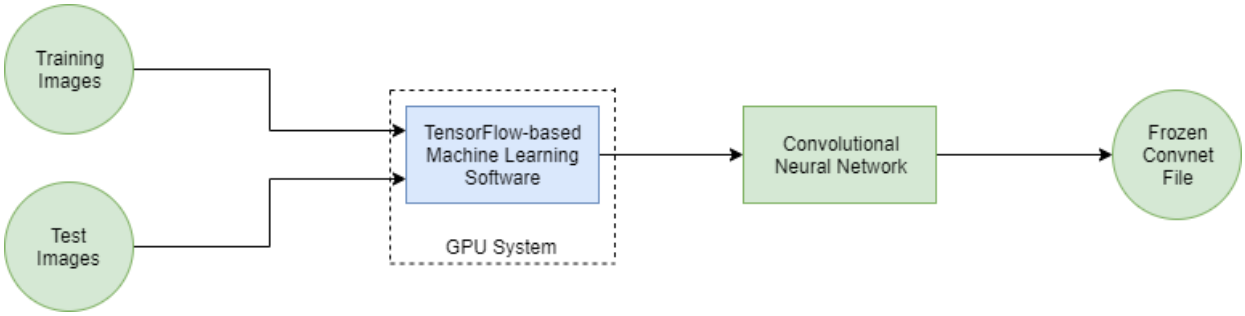

Fig 5. Diagram illustrating the process of training the neural network data model that will enable BinBot to recognize objects.

The machine learning algorithm will take as input approximately one thousand images for each waste objects we want to make BinBot's image recognition able to identify. Running on a GPU system and implementing Google's TensorFlow API, the software will train a convolutional neural network data model. There will also be several hundred images reserved as a test data set, to verify the neural network's success on data it was not trained on. This model can then be exported, a process called "freezing", to be implemented in the server-side OpenCV implementation.

MACHINE LEARNING

Model Training Test/Evaluation Metrics

During the course of the process of training the convolutional neural network model for image detection, several test metrics are output by the training software at every one-thousandth iteration of the training. In order to ensure the model is successful at object detection, it was crucial to monitor these metrics and adjust when needed.

The first main test metric for monitoring the training process is the "Average Precision", which gauges the accuracy of the training model at identifying the objects in the provided test set of images. Mathematically, it can be expressed as the total number of successful identifications divided by the sum of all identifications, successful or false.

The second main metric is the "Average Recall", which is a measure of how well the model can identify all expected objects in the test set. It can be mathematically expressed as the total number of successful identifications divided by the sum of all identifications made (correct and incorrect).

In early attempts at training, the average precision began to rise too quickly and plateau early. This resulted in a low-accuracy model, and it became apparent that the test set needed to be modified to be more challenging. The test set was augmented with more "difficult" images of smaller and further away objects, as well as more images containing several different objects.

Once these augmentations were made and the test set diverged more from the training images, the Average precision took much longer to reach a plateau of about 85% within about 200,000 training iterations, a much more realistic expectation. Following this, the resulting model performed much better in the manual testing phase.

COMPLETED MODEL MANUAL TESTING

At various intervals during the training process, I "froze" the training checkpoints for manual testing. "Freezing" refers essentially to converting the variables in the convolutional neural network model into constants and outputting it into a data file that can be loaded into other software to perform object identification.

For this step of the testing process, a simple python application was made to load the frozen model and augment the images from a PC webcam with identification bounding boxes and certainty percent scores. Using this, we could see in real time how the model performed at identifying objects and the number of false positives in quickly changing images in various environments. Using this we could also get a basic idea of how far away objects could be detected, and which objects the model was more successful with than others.

ROBOT KIT AND RASPBERRY PI

Build and Functionality Testing
Upon completion of assembling the robot kit, we wrote basic demo instructions for the robot to execute. These demo instructions would test the functionality of each mechanical component without the need of the real-time instructions generated by the processing server. This was done by manually constructing mock JSON files containing instructions in a structure reflecting that of the instructions generated by the server.

The mock instructions for the treads would test that the robot can move in each direction. This also tested the treads were capable of turning 360° in both directions and a specified distance.

Unit testing was done using Python's Unittest library. Each interface contained a unit test file that would be discoverable by the main unit test file. The main unit test file would discover all Unittest files within the project and run each test.

Arm Tests:
To calibrate the movements/motions of the arm/claw of the robot

Methods used to test the functionality of each arm component, once each component was tested separately, methods were put together in a specific sequence that would be called when waste needed to be pick up. Motions were observed until the right movements were executed.

def hand(command)
Expected: A string representing a command the arm would execute
This manual test was used to check whether the arm would move properly based on the command being called. The command varied between *'in'* which would call for the arm to move out and *'out'* which would call for the arm to bring back in the arm.

def openClaw()
Expected: Did not take in a parameter
This manual test was used to check whether the claw of the robot would open to the desire position when called. Various inputs had to be tested to get the desired width to open too.

def catch()
Expected: Did not take in a parameter
This manual test was used to check whether the claw of the robot would close to the desired position when called. Various inputs had to be tested to get the robot claw to completely close.

Distance Sensor Tests:

To check the distance an object is from the robot using an ultrasonic sensor

Method to use the ultrasonic sensor and return the distance an object is from the robot

def checkdistance():

Expected: Did not take in a parameter

A manual test was used along with this method where objects were put in front of the sensor and the distance would then be printed on the screen. The distance was then measured to make sure the outputs were accurate with what the distance sensor was returning. This test was used to make sure accurate distance was being computed.

Tread Tests:

Before implementing the functions to invoke the tread motors, a test function was made to verify instructions were being sent to the Tread interface correctly. The *test_executeTreadInstruction()* function replicated that of the actual instruction execution function but would only print to the terminal the action the treads would be taking instead. This would include the distance, speed, and motor duration time, etc. as shown below.

```
if angle == 0.0 or angle == 360.0:
    print("Moving " + str(distance * 10) + " cm forward.")
    print(f"Distance: {distance}\nSpeed: {speed}\nSleep: {distance * d_scale}")
```

Prior to integration with the processing server, the robot would execute hardcoded instruction sets to test the movement of the robot. These instructions included moving certain distances, as well as testing left and right turns at different angles; such as 45°, 90°, 180°, etc. Below shows a sample of a test instruction set.

```
sample_instructions = dict(treads=[
        {"angle": 90.0, "distance": 1.0},    # turn right 90*
        {"angle": 0.0, "distance": 0.775},  # move forward 7.75 cm
        {"angle": 45.0, "distance": 1.0},    # turn right 45 degrees 8 times
        {"angle": 45.0, "distance": 1.0},
        {"angle": 45.0, "distance": 1.0},
        {"angle": 45.0, "distance": 1.0},
        {"angle": 45.0, "distance": 1.0},
        {"angle": 45.0, "distance": 1.0},
        {"angle": 45.0, "distance": 1.0},
        {"angle": 45.0, "distance": 1.0}
])
```

Because the Tread interface interacts with physical components, most of the testing for BinBot's movement required the trial and error process. This included implementing different biasness scalars that would modify the speed and duration that the motors would move. This was necessary due to the robot having different levels of friction on different surfaces; causing it to slide at certain speeds. The robot would also turn at varying rates based on the surface as well. Another bias variable was introduced to scale the duration that the motors would run so that it would match the angle intended to turn. The following variables contain the calibrations which we found best for the classroom's floor.

```
d_scale = 0.3       # Scales sleep to unit of distance
speed = 100         # Speed at which the treads move
slide_bias = 0.85   # Scales the speed of the counter-turning tread based on friction
sleep_bias = 0.90   # Scales the sleep time based on friction of terrain
```

Camera Tests:

Prior to server integration, the Camera interface was tested by saving images to files on the Raspberry Pi. The files were saved in different formats to test the best way to send images to the server. Initially, the camera would capture an image and write it to a .jpg file saved on the Pi. We could then verify the camera was correctly taking images by viewing the file through WinSCP. The interface also included a test to camera an image and encoding it to a base 64 byte array, writing the array to a file as text. Using an online converter, we could verify the byte array could be decoded as the image. We then used this file on the data processing server to test decoding the file into an image on the server side. Once we completed the testing process, the interface was changed to capture images as a byte array into memory rather than having to read and write the contents to the disk.

Data Processing Server

The data processing server has been tested using a combination of unit tests and manual testing. Unit testing was mainly used to ensure algorithms and classes were functioning as expected in simple cases. For more complicated tasks and components, such as transmitting data over a network and preserving the integrity of that data, manual testing had to be used.

Data Transmission

Data transmission was tested almost entirely manually. By comparing data sent over from the source to data received by the destination. More details can be found in the integration tests

Instruction Calculation

Instructions Calculation was tested almost entirely manually as well. This means running the server and BinBot to see if it could travel to waste using instructions, then tweaking the instruction algorithm to increase accuracy of instructions for BinBot to follow such that it successfully navigates to the waste.

Unit Tests

For simpler tasks, such as ensuring data integrity or json conversions, unit tests were used. Below is a list of unit tests

Instruction

Instruction(Object[][] o)

Expected: An Instruction object properly created based on its input Object[][]
Test: instructionOTest()
This is tested by inputting a very controlled object[][] and then calling the json() method, which returns a json string version of the Instruction object and comparing the resulting json string with the expected json string. If the two are matching, then the test has passed.

Instruction(String json)

Expected: An Instruction object properly created based on its input string
Test: instructionJTest()
This is tested by inputting a very controlled json string and then calling the json() method, which returns a json string version of the Instruction object and comparing the resulting json string with the original json string used to construct

the object. If the two are matching, then the test has passed. Note, the json string should never change, so this ensures the input json string is being read correctly.

String json()

Expected: a json string that properly represents the Instruction it was called from
Test: jsonTest()
This is tested by inputting a very controlled json string and then calling the json() method, which returns a json string version of the Instruction object and comparing the resulting json string with the original json string used to construct the object. If the two are matching, then the test has passed. Note, the json string should never change, so this ensures the input json string is being read correctly. In addition, this test is exactly the same as the above test but included in that case that our implementation changes in the future.

TreadsInstruction

List<Pair<Double, Double>> calcInstructions(double x, double y, double w, double h)

Expected: a list of double pairs, each representing an angle and then a distance that the BinBot robot should move
Test: calcInstructionsTest()
This is tested by inputting a set of parameters, which represent the location and dimensions of a waste object and testing to see if the resultant list of instructions matches our expected list of instructions.

ArmsInstruction

List<Double> calcInstructions(double x, double y, double w, double h)

Expected: a list of doubles each representing an angle that it's corresponding joing should rotate
Test: calcInstructionsTest()
This is tested by inputting a set of parameters, which represent the location and dimensions of a waste object and testing to see if the resultant list of instructions matches our expected list of instructions.

ANDROID MOBILE APPLICATION

The mobile application was created to be able to send signal information to the server. The signal (Start or Stop) are the states the user wants the robot in. To ensure that when the user hits the start or stop button, the correction information will be sent. I created a dummy server that works of a socket and receives a string which displays to console.

Start_button:
Start a server socket
Create an object
Covert object to a string (with bool true)
Send json as a string to server

Stop_button:
Create an object
Convert object to a string (with bool false)
Send json string to server

After implementing the functionality of these buttons and the dummy server printed out the corrected string the application worked as supposed.

Mobile Server Communication

INTEGRATION TESTS

All integration testing was done manually by using the various components together and observing the outcome. Tests were considered to be passed if the BinBot system and components performed as expected.

Server Robot Communication

Ensuring successful communication was done by creating a mock json string and attempting to send it across the socket connection both ways. If the json string was successfully sent and received as it was sent, the test was considered to have passed.
Furthermore, conversion from the json string to a POJO was tested as well, and if successful, considered to have passed the test. The basic test followed the below format:

Server:
Create object o
Convert o to json string so
Send json string so to Robot
Receive json string sj
Convert sj to object j

Robot:
Receive json string so
Convert so to object o
Convert o to json string sj
Send json string sj to Server

Server Mobile Application Communication

Ensuring successful communication was done by creating a mock json string and attempting to send it across the socket connection both ways. If the json string was successfully sent and received as it was sent, the test was considered to have passed.
Furthermore, conversion from the json string to a POJO was tested as well, and if successful, considered to have passed the test. The basic test followed the below format:

Server:
Create object o
Convert o to json string so
Send json string so to Robot
Receive json string sj
Convert sj to object j

Mobile Application:
Receive json string so
Convert so to object o

Convert o to json string sj
Send json string sj to Server

Instruction Execution
Ensuring instructions were properly executed was done by sending mock instructions from the server to BinBot and having BinBot execute those instructions. This also required successful Data transmission and json conversion. If BinBot moved in the manner dictated by the mock instructions properly, it was considered to have passed the test.

Server:
Loop:
Create instruction I
Convert I to json j
Send json j to Robot
Wait for a response
Robot:
Loop:
Wait to receive json string j
Convert json string j to Instruction I;
Execute Instruction I
Send acknowledgement to server

Mobile Application Server Operation Toggle
Testing the Mobile applications ability to toggle the server was done by manually clicking the button to start and stop the server and having the server print its status in a loop. If the server

successfully printed the “stopped” status when appropriate and the running status when appropriate, it was considered to have passed the test.

Server:
    Thread 1:
        Loop:
            If (stopped) print(stopped)
    Thread 2:
        Loop:
            Receive json string s from app
            Convert string s to message m
            Set boolean stopped to boolean in message m
Mobile App:
    Upon clicking button:
        Send message with stopped/start boolean in it to server as json

ACCEPTANCE TESTS

For acceptance tests, we had one main test, because we had really only one main user experience. The test was simply laying assorted waste around BinBot. Placing BinBot in this environment, turning on BinBot and running the proper program with the server IP, then starting the server. If BinBot goes around retrieving waste, then the test is considered passed. Supplementary to this main test, during the main test we test the mobile application by connecting it to the server, then clicking start and stop and checking if BinBot appropriately starts and stops. If it does, the test is considered passed.