BinBot

Test Procedures

REVISION HISTORY

Revision #	Author	Revision Date	Comments
0.1	Sean Reddington	October 17, 2019	Initial template
1.0	Kwamina Thompson	October 20, 2019	Android Mobile App
1.1	Sean DiGirolamo	October 20, 2019	Data Processing Server
1.2	Jose Silva	October 20, 2019	Robot Kit and Arduino
1.3	Sean Reddington	October 20, 2019	Robot Kit and Arduino, System Overview, Formatting
1.4	Michael Savitski	October 21, 2019	Machine Learning

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 and place it in a waste bin attached to the robot. The image feed from BinBot, with additional visual indicators of identified waste, can be transmitted to a mobile application for observing BinBot's progress and success.

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 an Arduino Mega 2560 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 receive images from the data processing server which will allow the user to watch a live feed of BinBot's camera view. 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 such as the estimated distance and status messages like "traversing" or "collecting".

System Block Diagram

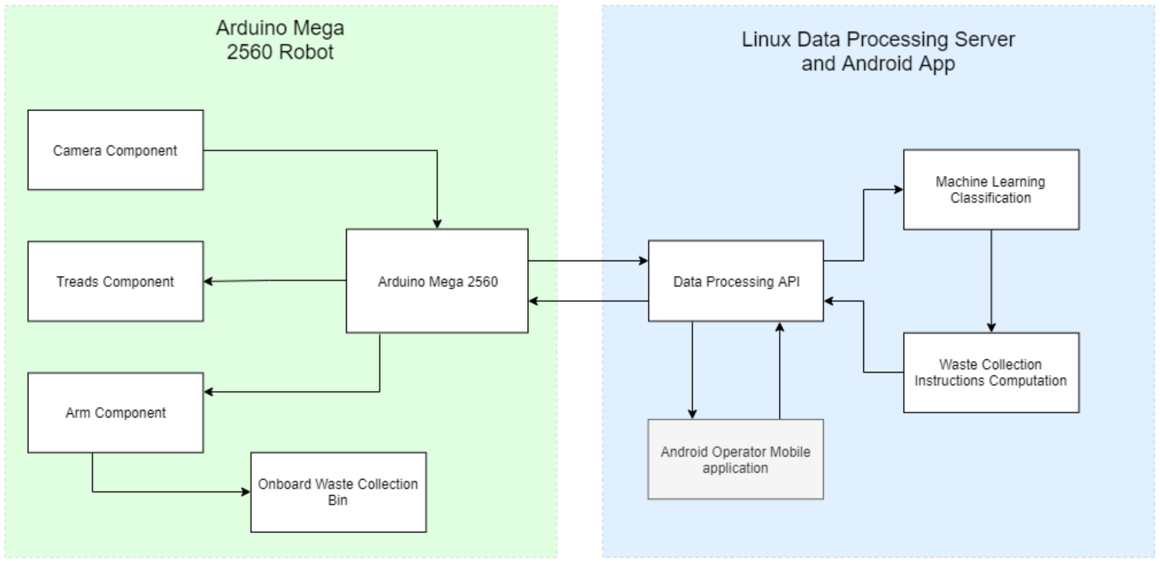
Arduino Mega 2560 Robot
Linux Data Processing Server and Android App
Camera Component
Treads Component
Arduino Mega 2560
Arm Component
Onboard Waste Collection Bin
Data Processing API
Machine Learning Classification
Waste Collection Instructions Computation
Android Operator Mobile application

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 Arduino Mega 2560. The Arduino Mega 2560 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; passing the results to the Android operator mobile application. 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 Arduino board. The Arduino will then pass the instructions to the tread and arm interfaces to collect the object, placing it into an onboard waste collection bin.

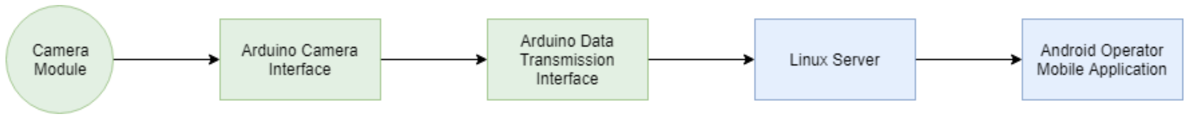
Camera Module
Arduino Camera Interface
Arduino Data Transmission Interface
Linux Server
Android Operator Mobile Application

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 Arduino'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.

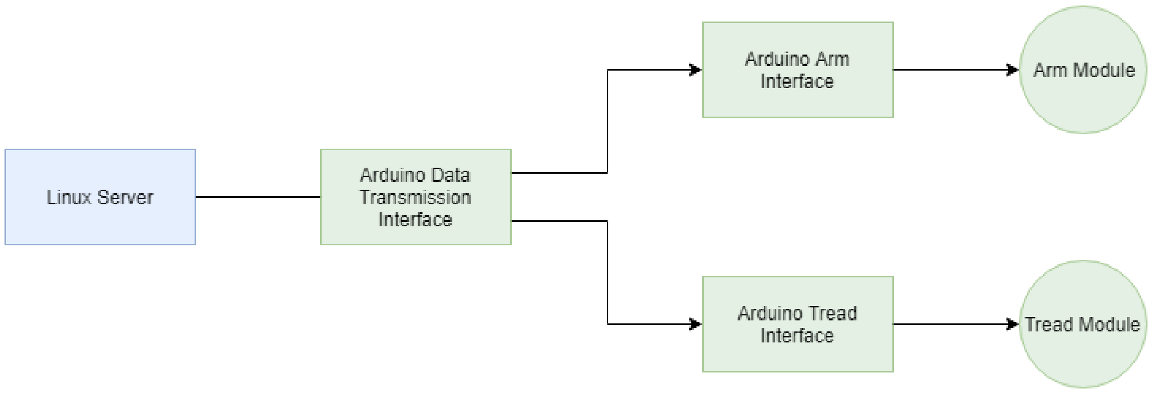

Fig 3. Diagram displaying interaction between the Linux Server and the Arduino's interfaces in order to operate the robot.

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

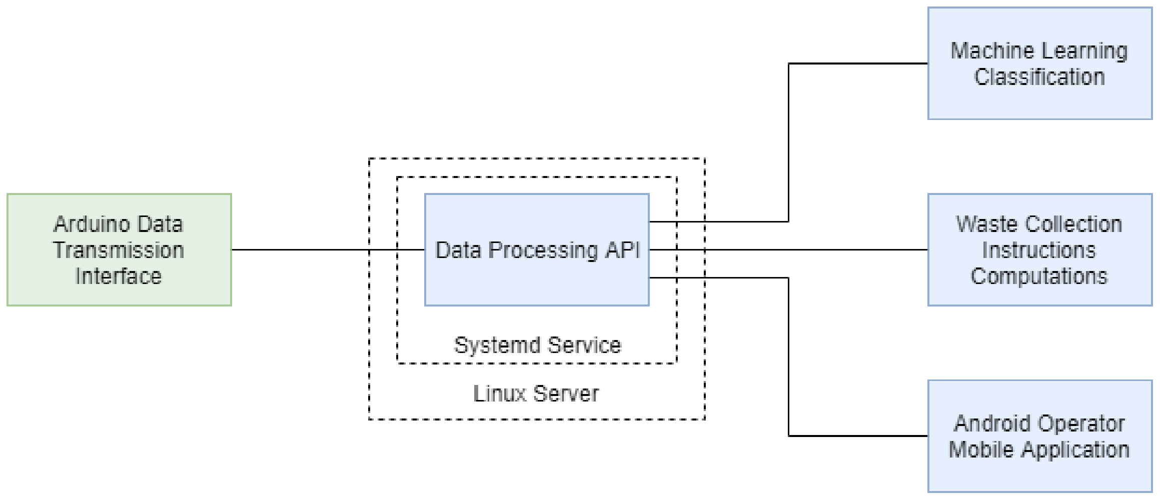
Machine Learning Classification
Arduino Data Transmission Interface
Data Processing API
Systemd Service
Linux Server
Waste Collection Instructions Computations
Android Operator Mobile Application

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

The Arduino 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 Systemd 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. The image including the results of the object classification will also be sent to the Android mobile application.

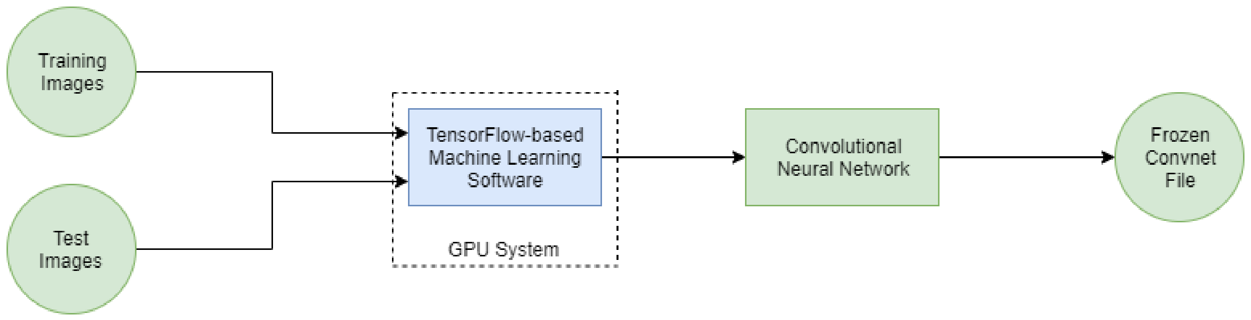
Training Images
Test Images
TensorFlow-based Machine Learning Software
GPU System
Convolutional Neural Network
Frozen Convnet File

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 trash 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

Unit Tests

Python provides unit testing functionality similar to JUnit, which will be used to test both the public and private methods used in the software after changes to the code, and before regular use.

Main Class Tests

def test_args(): Tests the parse_agrs() method by passing in a line of text and validating the parsed arguments are as expected.

def test_init(): Tests the initialize_convnet() method by calling it with a set of parameters and a path that will not modify any existing saved model, and confirm the model is valid and the saved model file exists.

TrainConvnet Class Tests

def test_load_images(): Passes in the path to a set of images in the project directory intended for testing, and verified that the returned ImageGenerator object's parameters are correct.

def test_training(): Passes in a defined model and generator (using the afore mentioned test directories) and validated the returned training history object has the correct expected parameters.

FreezeModel Class Tests

def test_freezing(): Passes in a directory path intended for testing and a path to saved model files created for testing, which will not over-write or use any existing model files, and validate that the file exists.

Manual Tests

For the sake of validation testing, a number of manual tests are to be built into the training software to validate the software works correctly, and that training has actual results.

Graphs

At the completion of model training, two graphs are always generated from the history object and displayed. The first is a plot of the percent success rate of the training set images versus the test set images. The second is a plot of the loss on the testing images versus the lost on the training images. The loss is the result of a built-in loss function, the value of which reflects the deviation of the actual training results from the expected training results.

Image Validation
There will be a simple optional method, called by a command line argument, which will display the images to be used out to the user before (or instead of) beginning training to validate they were read in correctly, and that the intended data set was used.

Actual Implemented Convnet Validation
The best to determine the exported frozen convnet is usable after training is to use that file in a simple OpenCV implementation. This acceptance testing software will simply capture a steady stream of images from the local computer's webcam and display those images to the use with bounding boxes drawn around detected objects.

Robot Kit and Arduino

Build and Functionality Testing

Upon completion of assembling the robot kit, we plan to write basic demo instructions for the robot to execute. These demos would test the functionality of each mechanical component without the need of the real-time instruction generated by the processing server. This will be 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 would also test that the treads are capable of turning 360° in both directions. The instructions for the arm would test the full range of motion for each of the 4 joints. We will also include a test for spinning horizontally at the base of the arm.

Movement Correction

Another requirement that must be tested for correcting the distances that each component moves. Because the Arduino moves the mechanical components by simply applying power to the connected pin, we must obtain the correct power output to translate to a measurement of distance. This includes finding the correct amount of power to move the treads forward in 1 meter increments as well as turning the robot in increments of about 5°. Correcting the range of motion to a measurement applies for the arm joints as well.

Treads Tests: To calibrate the car direction with a calibration value

```
Void forward_test(int value){
    Switch(value)
        Case 1: Motor_Forward
        Default
Void back_test(int value)
    Switch (value)
        Case 2: Motor_Back
        Default
Void leftTurn_test(int value)
    Switch(value)
        Case 3: Motor_Left
        Default
Void rightTurn_test(int value)
    Switch(value)
        Case 4: Motor_Right
        Default
Void stop_test(int value)
```

```
        Switch(value)
                Case 5: Motor_Stop
                Default
```

Each method expects a value which will calibrate the direction the car will move in based on that value; each method will have to tested by implementing them directly on the robot and observing that BinBot moves in the correct direction based on our command.

Arm Tests: To calibrate the arm direction with a calibration value

```
    Void shoulder_test(int value){
        Switch(value)
                Case 1: Shoulder_motor
                Default
    Void elbow_test(int value)
        Switch (value)
                Case 2: Elbow_motor
                Default
    Void wrist_test(int value)
        Switch(value)
                Case 3: Wrist_motor
                Default
    Void clawOpen_test(int value)
        Switch(value)
                Case 4: Claw_motor
                Default
   Void clawClose_test(int value)
        Switch(value)
                Case 5: Claw_motor
                Default
```

These methods will used to test the functionality of the robot, like the treads tests each method will have to called once uploaded onto BinBot individually. BinBot will be observed that each movement of the arm corresponds with the command being called.

Camera Tests:

Void capture_photoTest()

Test: This method will be used to test the functionality of the camera module on BinBot. This method will activate the camera on BinBot and take a photo, that photo will then be saved to a file on the PC. We will then inspect the photo to make sure that it captured image clearly and correctly.

get_proximityTest()
Test: This test will be used to check if BinBot is able to acquire the proximity of an object. Using the camera module on BinBot it will get an image of the object and state True/False if it is able to get the proximity of that object and return the actual proximity of the object. This test will have to run directly using BinBot's camera.

in_rangeTest()
Test: This test will be used to check if the object identify is in range for BinBot to pick up with its claw. Will return True/False if object is/is not in range. We will observe the results and have to verify if the object in BinBot's view is in range or not to check the accuracy of this method.

Data Processing Server

The BinBot server will use standard Test-Driven Development for testing its functionality and software rigidity. Each module/class in the architecture will have its own associated test class to perform tests on the functionality of that module. For each individual function, a test is created to ensure that it returns the expected output when given a controlled input.

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.

BotConnection

BotConnection(int port)
void accept()
void sendToBot(String s)
String receive()
Expected: the same String that was sent to the BinBot robot, returned to the server through its own test.
Test: testConnection()
Because of the nature of connections, this test must be performed in coordination with the BinBot robot, to validate that data can be sent, and data can be received. The server starts by establishing a connection with BinBot, then sending a controlled String to BinBot. It then waits to receive the same String back. If at any point the connection is not established, the String is not sent, or the String is not returned, the test will fail.

AppConnection

AppConnection(int port)
void accept()
void sendToApp(String s)
String receive()
Expected: the same String that was sent to the BinBot mobile application, returned to the server through its own test.
Test: testConnection()
Because of the nature of connections, this test must be performed in coordination with the BinBot mobile application, to validate that data can be sent, and data can be received. The server starts by establishing a connection with the mobile application, then sending a controlled String to it. It then waits to receive the same String back. If at any point the connection is not established, the String is not sent, or the String is not returned, the test will fail.

ANDROID MOBILE APPLICATION

Android App will be implementing Test Development Testing
Testing: onJSONRECIEVER{}
To test this method/class, I am planning on creating a hard coded json file. This file is going to contain image files which will be converted into bytes and stored in the json file. Furthermore, we plan to use myJSON online API to simulate the JSON file been hosted on a server so that all internet connections edge cases could be come up.