Laura Curioso
Dec 21 2017

Project 3: Perception Writeup

Project Overview

For perception, three exercises were completed for analyzing the image for object recognition by first segmenting the image, clustering the segmented data, and then training the program to recognize the objects. These steps were then implemented in a new environment with three different test worlds where a PR2 robot recognizes the objects, picks them up, and then places them into bins.

Perception Pipeline

In addition to using code from exercises 1-3, a new ros node was created and the subscriber was changed. New data is from the RGB-D camera and the new objects from worlds 1-3 need to be trained for the PR2.

Reading in the Data

In order to read in the data, we needed to modify the ros node to subscribe to the PR2 world point cloud 2 topic:

```
# ROS node initialization
rospy.init_node('recognition', anonymous=True)

# Create Subscriber to receive the published data coming from the
#   pcl_callback() function that will be processing the point clouds
pcl_sub = rospy.Subscriber('/pr2/world/points', pc2.PointCloud2,
                           pcl_callback, queue_size=1)
```

The pcl_callback function is then called every time the sensor publishes a new pc2.PointCloud2 message.

Statistical Outlier Filtering

Since we know the exact number of objects, we can utilize a function called K-means to cluster pixel data into similar groups. From the exercise 1 and after playing with the settings, I set the cluster size to 20 and the threshold to 0.1.

```
# Statistical outlier filtering
outlier_filter = cloud.make_statistical_outlier_filter()
# Set the number of neighboring points to analyze for any given point
outlier_filter.set_mean_k(20)
# Any point with a mean distance larger than global will be considered out
outlier_filter.set_std_dev_mul_thresh(0.1)
cloud_filtered = outlier_filter.filter()
```

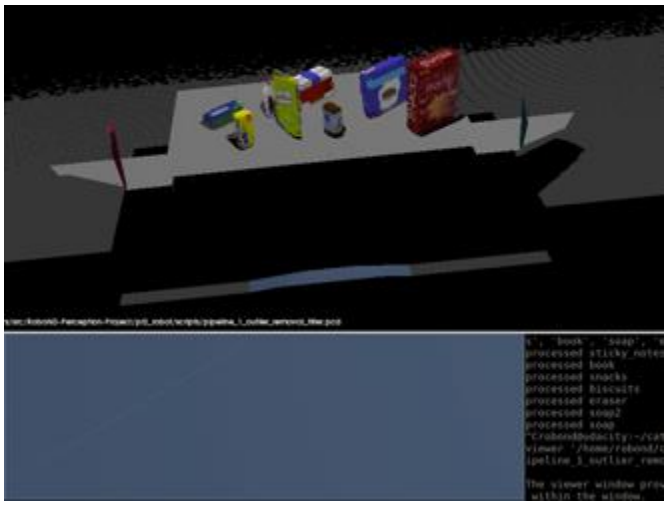

Figure 1: K-means Output

Exercise 1: Filter out Noise using RANSAC.py

Step 1 was to filter out noise around the objects by performing segmentation on the point cloud data. The first filter used is voxel grid downsampling which can be found in the RANSAC.py. A leaf size of 0.01 seemed sufficient for the filtered output. I didn't seem to noisy so I left it alone

```
# Voxel Grid Downsampling
vox = cloud_filtered.make_voxel_grid_filter()
LEAF_SIZE = 0.01
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
cloud_filtered = vox.filter()
```

The result can be viewed in the PCL viewer (Figure 2).

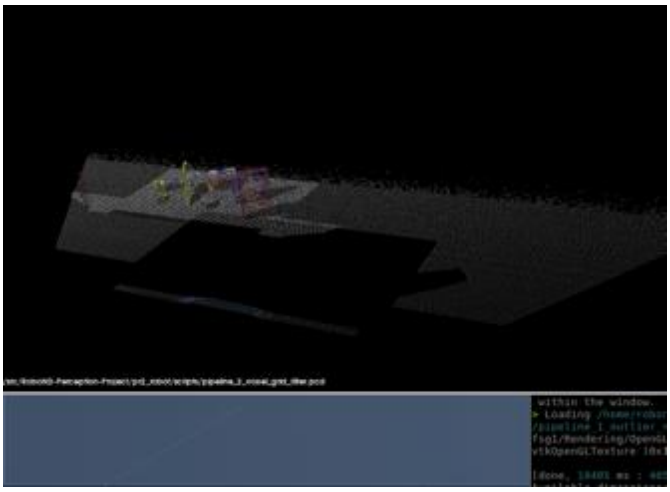

Figure 2: Voxel Grid Downsampling for World 3

After the image is downsampled, pass through filtering was applied to "crop out" the unwanted noise around the objects of interest. For the pass-through values, I found setting the filter axis minimum to 0.5 and the axis maximum to 1.5 generated the following result in Figure 3 below:

```
# Create a PassThrough filter object.
passthrough = cloud_filtered.make_passthrough_filter()

# Assign axis and range to the passthrough filter object.
filter_axis = 'z'
passthrough.set_filter_field_name (filter_axis)
axis_min = 0.5 #.765
axis_max = 1.5 #1.3
passthrough.set_filter_limits (axis_min, axis_max)
cloud_passthrough = passthrough.filter()
passthrough = cloud_passthrough.make_passthrough_filter()

filter_axis = 'y'
passthrough.set_filter_field_name(filter_axis)
axis_min = -0.5
axis_max = 0.5
passthrough.set_filter_limits(axis_min, axis_max)

# Finally use the filter function to obtain the resultant point cloud.
cloud_filtered = passthrough.filter()
```

Figure 3: RANSAC Function Output for World 3

RANSAC plane fitting provides further segmentation by removing the object background data since we have prior knowledge of a certain shape being present. The table is modeled as a plane and by identifying the inliers vs. outliers, you can remove the table itself resulting in only the objects of interest being displayed in Figure 4 below.

Figure 4: Inliers

The ExtractIndices Filter allows you to extract points from a point cloud by providing a list of indices. After the RANSAC plane fitting, the output inliers corresponds to the point cloud indices that were within max distance of the best fit model. This allows for the objects then to be “extracted” and can be viewed in the extracted_outliers.pcd.

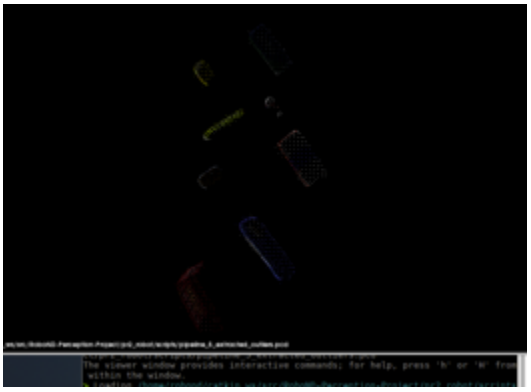

Figure 5: Extracted Outliers (Objects of Interest)

Exercise 2: Clustering and Segmentation

Step 2 was to apply clustering to the points using an Euclidean (DBSCAN) algorithm. Even though the k-means algorithm is used, DBSCAN doesn't require the specific number of clusters to be set but do know the density of the cluster by setting the max distance between points within that cluster. DBSCAN also proves more useful if you want to remove outliers from your cluster data. This cluster data is thein outputted to the ros core. The max distance was kept the same as in the exercise as 0.01.

```
# Create the segmentation object
seg = cloud_filtered.make_segmenter()

# Set the model you wish to fit
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)

# Max distance for a point to be considered fitting the model
# Experiment with different values for max_distance
# for segmenting the table
max_distance = 0.01
seg.set_distance_threshold(max_distance)

# Call the segment function to obtain set of inlier indices and model coefficients
inliers, coefficients = seg.segment()
```

For the Euclidean parameters, the start filter of a cluster tolerance of .015, min cluster size of 20, and max cluster size of 1500 was barely enough to teach the model for world 1 but not for worlds 2 and 3. After a few trials, I came up with a tolerance of .05, min cluster size of 100, and max cluster size of 3000 to extract the appropriate amount cluster indices.

```
# Create a cluster extraction object
ec = white_cloud.make_EuclideanClusterExtraction()
# Set tolerances for distance threshold
# as well as minimum and maximum cluster size (in points)
# NOTE: These are poor choices of clustering parameters
# Your task is to experiment and find values that work for segmenting objects.
ec.set_ClusterTolerance(0.05)#.015
ec.set_MinClusterSize(100)#20
ec.set_MaxClusterSize(3000)#1500
# Search the k-d tree for clusters
ec.set_SearchMethod(tree)
# Extract indices for each of the discovered clusters
cluster_indices = ec.Extract()

# TODO: Create Cluster-Mask Point Cloud to visualize each cluster separately
#Assign a color corresponding to each segmented object in scene
cluster_color = get_color_list(len(cluster_indices))
```

Exercise 3: Features Extraction and Object Recognition

Using SVM, a model was trained to capture features to help identify objects. HSV was then implemented to enhance the accuracy of object recognition since some of the objects had similar color features. Typically, the more poses that are implemented, the better the accuracy of the model fit. Accuracy is

displayed in a confusion matrix. Increasing the bin size also enhanced the accuracy of the model fit. After starting off with 100 poses, in order to get accurate predictions for worlds 2 and 3, I had to increase the amount of poses to 1000 to ensure that the model was taught to an accuracy of at least 70%.

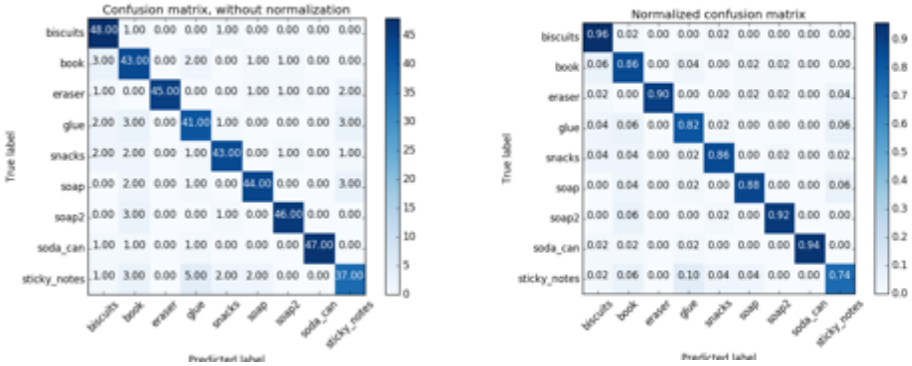

Figure 6: 9 Objects SVM Confusion Matrix (Trial N)

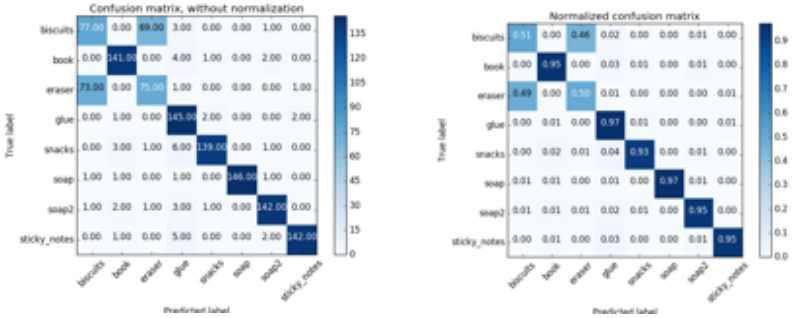

Figure 7: 9 Objects SVM Confusion Matrix (Trial 1)

Changing the Launch Code

The launch code had to be changed to load the correct world and pick list:

```
<!--Launch a gazebo world-->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <!--TODO:Change the world name to load different tabletop setup-->
  <arg name="world_name" value="$(find pr2_robot)/worlds/test2.world"/>
  <arg name="use_sim_time" value="true"/>
  <arg name="paused" value="false"/>
  <arg name="gui" value="true"/>
  <arg name="headless" value="false"/>
  <arg name="debug" value="false"/>
</include>

<!--TODO:Change the list name based on the scene you have loaded-->
<rosparam command="load" file="$(find pr2_robot)/config/pick_list_2.yaml"/>

<!--launch rviz-->
```

The PR2 environment was then launched and the trained model was used to identify the 9 world 3 objects. The result is displayed in Figure 7 below:

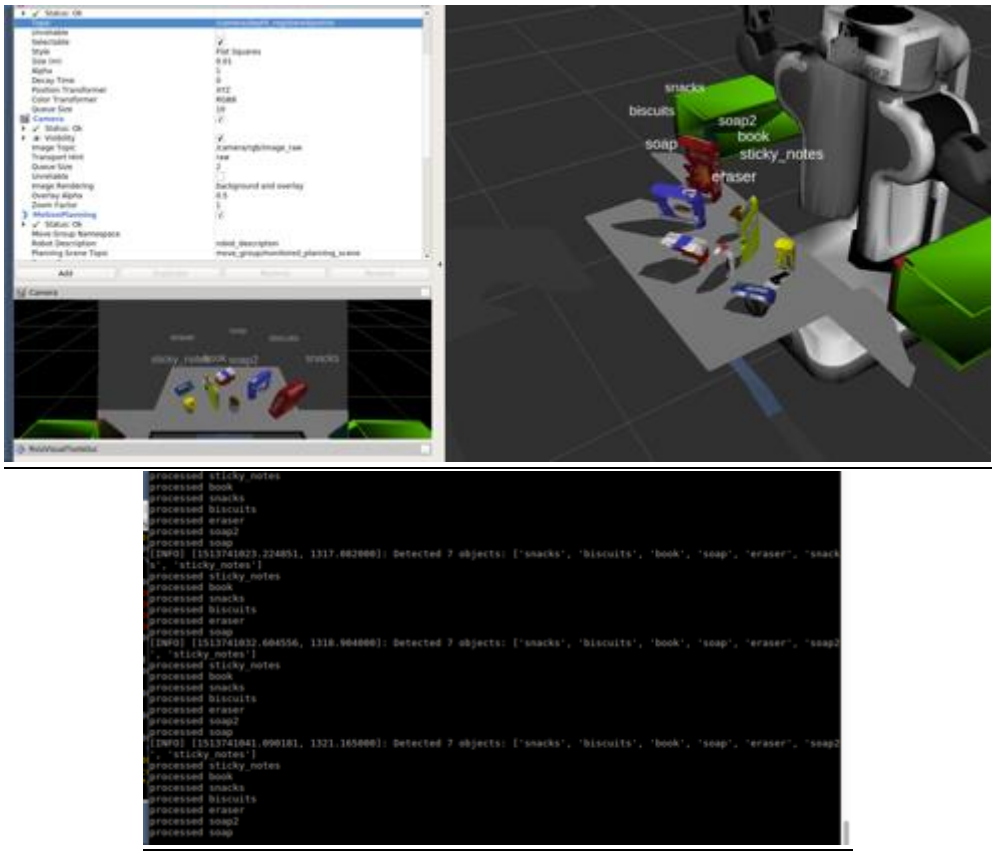

Figure 8: Object Recognition for World 3, Picklist 3

Object Recognition Project Results

I took snippets of the different topic views in world 3 to show that the code was publishing properly. The output yaml files for all 3 worlds are in the github repository.

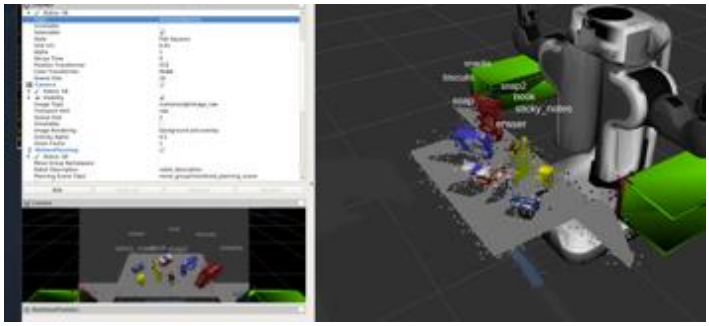

Figure 9: /pr2/world/points Topic View

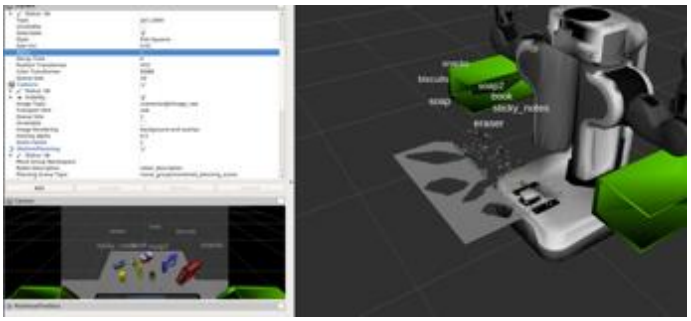

Figure 10: /pcl_table View

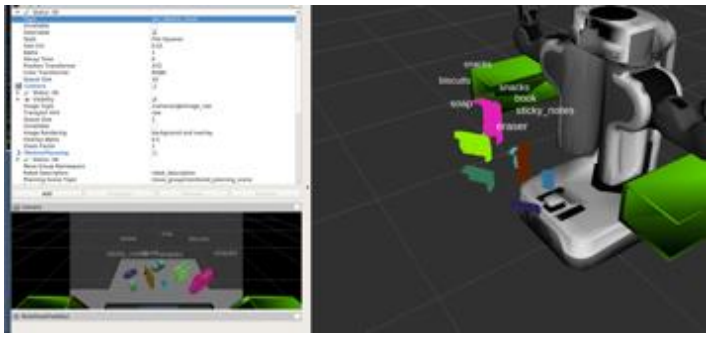

Figure 11: /pcl_objects_cloud View

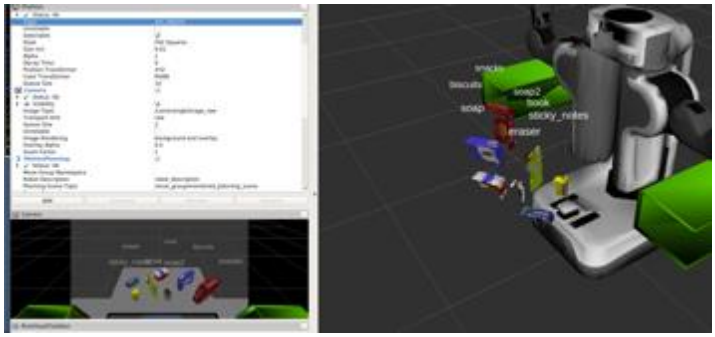

Figure 12: /pcl_objects View

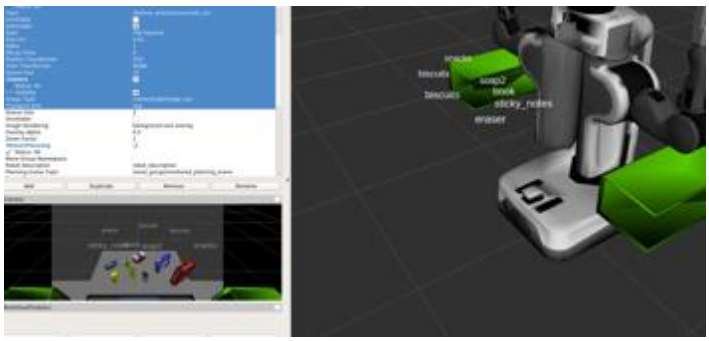

Figure 13: /feature_extractor/normals_out

Figure 14: World 1 Object Recognition

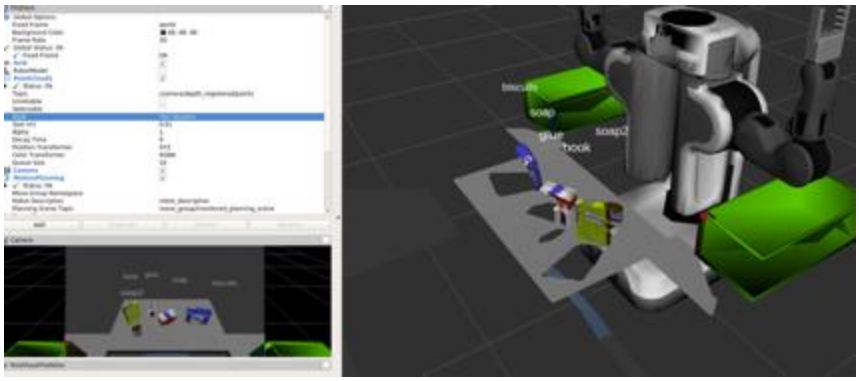

Figure 15: World 2 Object Recognition

Pick and Place ROS Message Request

To read the object list and dropbox list parameters, the following code was implemented:

```
# TODO: Get/Read parameters (from output yaml notes)

object_list_param = rospy.get_param('/object_list')
dropbox_list_param = rospy.get_param('/dropbox')
```

The objects are then converted into dictionaries:

```
        #centroid list:
        centroids = []
        #label list: dict_list
        output_list = []
        #dictionary list (yaml files dict)
        dict_list = []

    # TODO: Parse parameters into individual variables
    dropbox_name = {}
    dropbox_position = {}
    label_dict = []
```

From the object list parameter set, I was able to determine the number of objects in the set by extracting the length.

```
#Define number of objects:
num_of_objects = len(object_list_param)
```

This length check is then used to match the object pick list:

```
#match list length to parameter list length
if not len(object_list) == len(object_list_param):
rospy.loginfo("Object list does not match object pick list.")
return
```

Test scene initialization needs to be implemented and the data type needs to be set to a 32 float:

```
#initialize test scene:
test_scene_num = Int32()
num_scene = 3
test_scene_num.data = num_scene
```

For pick list logic, we cycle through the pick list labels and append the detected objects to the detected object list:

```
# TODO: Loop through the pick list
sorted_objects = []
# Convert the numpy float64 to native python floats
for i in range(num_of_objects):
    pick_list_label = objects[i]['name']

    for detected_object in detected_objects:
        if detected_object.label1 == pick_list_label:
            sorted_objects.append(detected_object)

            # Remove current object
            detected_objects.remove(detected_object)
            break
```

We then calculate the centroid positions and append it to the centroid dictionary:

```
dropbox_groups = []
for sorted_object in sorted_objects:

    # Calculate the centroid
    center_points = ros_to_pcl(sorted_object.cloud).to_array()
    centroid = np.mean(center_points, axis=0)[:3]

    # Append centroid as <numpy.float64> data type
    centroids.append(centroid)
```

Before we pick a pose to have the arms move or have the PR2 identify the dropbox locations and associate them with a color bin:

```
# TODO: Rotate PR2 in place to capture side tables for the collision map
#Define left (red drop box) and right (teal dropbox) positions from parameter list
red_dropbox_position = dropbox[0]['position']
teal_dropbox_position = dropbox[1]['position']
```

We then declare the arm name as a string and assign the object groups to the arm name data based on the bin color:

```
# TODO: Assign the arm to be used for pick_place (copy code from capture_features.py)

arm_name = String()
if obj['group'] == 'red':
        arm_name.data = 'left'
    elif obj['group'] == 'green':
        arm_name.data = 'right'
    else:
        print "ERROR, group must be Red or Green!"
```

We also have to get the point cloud data for the given object and assign the object name and group to the dropbox names/groups.

```
    # TODO: Get the PointCloud for a given object and obtain it's centroid
for j in range(len(sorted_objects)):

    object_name = String()
        #object_name = object_list_param[i]['name']
        object_name.data = obj['name']
        #object_group = object_list_param[i]['group']
    object_group = dropbox_groups[j]
```

We also append the centroid calculation and convert it into the proper 64 float type data:

```
#convert to 64 float data type:
centroids.append(centroid)
np_centroid = centroids[j]
    scalar_centroid = [np.asscalar(element) for element in np_centroid]
```

To create the pick pose we need to set the x,y,z positons to a scalar coordinate system and then assign that pose to the arm name data. The place pose position is then assigned to the corresponding arm name data:

```
#Create Pick Pose:
pick_pose = Pose()
    pick_pose.position.x = scalar_centroid[0]
    pick_pose.position.y = scalar_centroid[1]
    pick_pose.position.z = scalar_centroid[2]

# TODO: Create 'place_pose' for the object
#Assign pose positions to arm name data matrix:

#pose()
place_pose.position.x = dict_dropbox[arm_name.data][0]
    place_pose.position.y = dict_dropbox[arm_name.data][1]
    place_pose.position.z = dict_dropbox[arm_name.data][2]
```

We then assign all this data to the yaml dictionary and then output that data to the yaml file.

```
# TODO: Create a list of dictionaries (made with make_yaml_dict()) for later output to yaml format
yaml_dict = make_yaml_dict(test_scene_num, arm_name, object_name, pick_pose, place_pose)
    dict_list.append(yaml_dict)

# TODO: Output your request parameters into output yaml file

yaml_filename = "output_" + str(test_scene_num.data) + ".yaml"

send_to_yaml(yaml_filename, dict_list)
print "yaml file created successfully"
```

In the if__name__ == '__main__': function the ros nodes, subscribers, and publishers are created. Messages are then published using the rospy.publisher function to the proper pointcloud2 format.

Conclusion

For worlds 1,2, and 3, the classifier correctly identified the objects 100%. I did see some mislabeling in the confusion matrix so there is a probability that an object may be miss-classified. The probability score ranged from 0.74 up to 0.96. I chose this model over the first trial as it had a more consistent score vs. trial 1 where some of the scores were higher but the misclassification count was higher for biscuits and eraser. I did see on one of my initial runs that the soap2 was misclassified as snacks which is another reason I chose to improve the model.

Improvements to this model would be to get more data and more poses. Perhaps more statistical manipulation to remove noise, such as changing the cluster bins and applying some techniques used in deep learning would help the robot perceive the objects better. Object data of those objects in different colors might also help train the SVM model better.