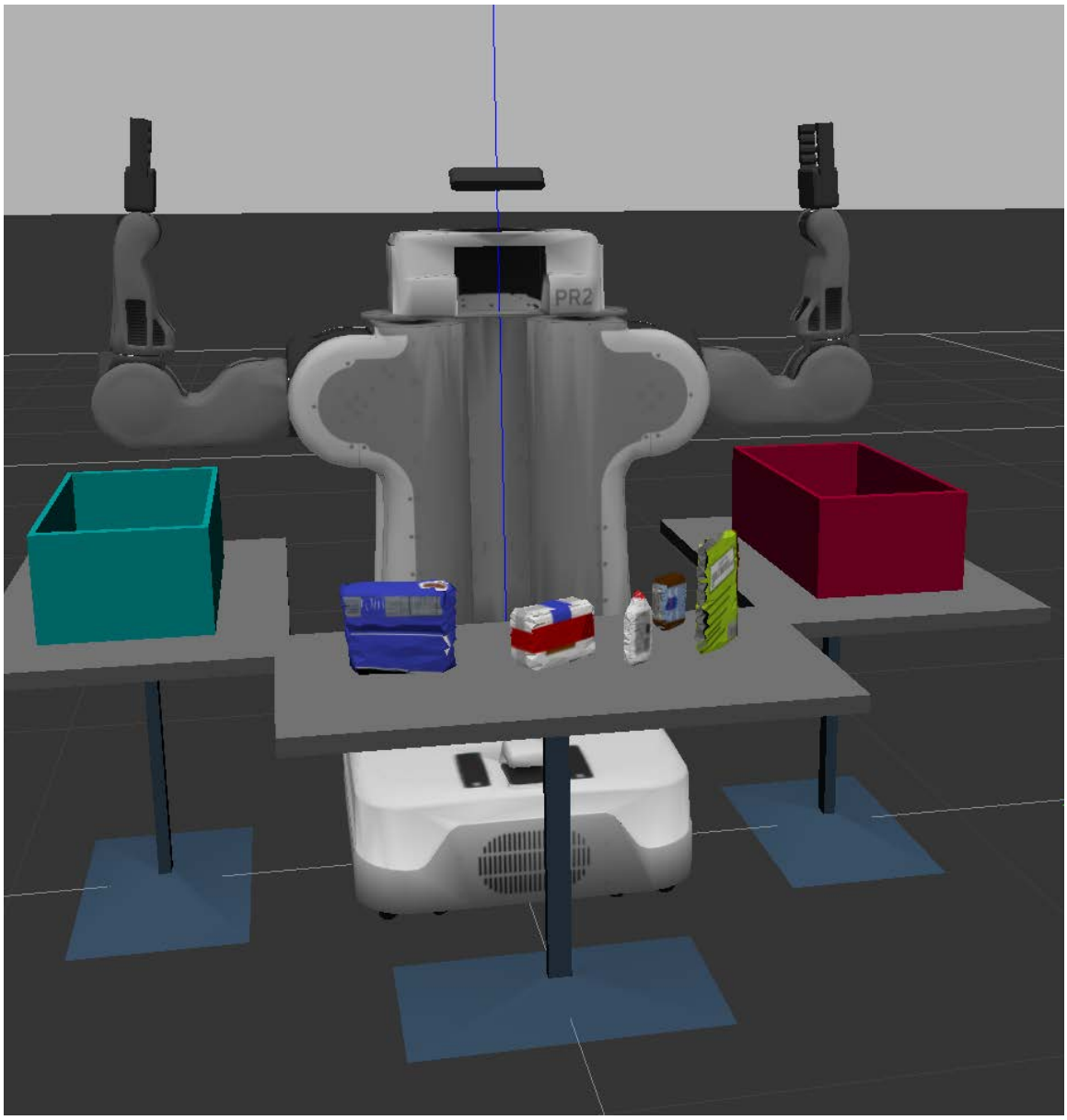


Project 3: 3D Perception

3D PERCEPTION WITH MACHINE LEARNING

Christopher J. Rukas | Robotic Software Engineer | 12/06/2018

Project Description

Project 3: 3D Perception is a recreating of an Amazon competition in which a robot must correctly identify objects and place them into the correct bins without human input. In this course/project a Support Vector Machine (SVM) is trained to identify an assortment of items. To achieve the identification, clusters of point cloud data are segregated from a the raw point cloud data. Pass through filters are applied to segregate data, RANSAC is applied to identify a planar surface, and Euclidean Clustering (DBSCAN) is used to identify individual objects. The robot then creates a motion plan to pick and place each object, in the desired order, into the appropriate bin.

Exercise 1 – Tabletop Segmentation

Exercise 1 requires identifying the tabletop data and separating objects from the tabletop. This is done in several steps:

1) Downsample the point cloud by applying a Voxel Grid Filter
2) Apply a Pass Through Filter to isolate the table and objects
3) Perform RANSAC plane fitting to identify the table's points.
4) Extract indices of the plane from the objects and create two separate clouds.

Voxelizing data points

```
# Voxel Grid filter
# Create a VoxelGrid filter object for input point cloud
vox = cloud_outlier_filtered.make_voxel_grid_filter()
# Choose a voxel (aka leaf) size
# Note: 1 is a poor choice of leaf size
LEAF_SIZE = 0.01
# Set the leaf size
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
# Call the filter function
cloud_vox_filtered = vox.filter()
```

A pass through filter identifies only data points within a specific zone with respect to the camera. In the following filter, two planes are created and only datapoints between those two planes are kept.

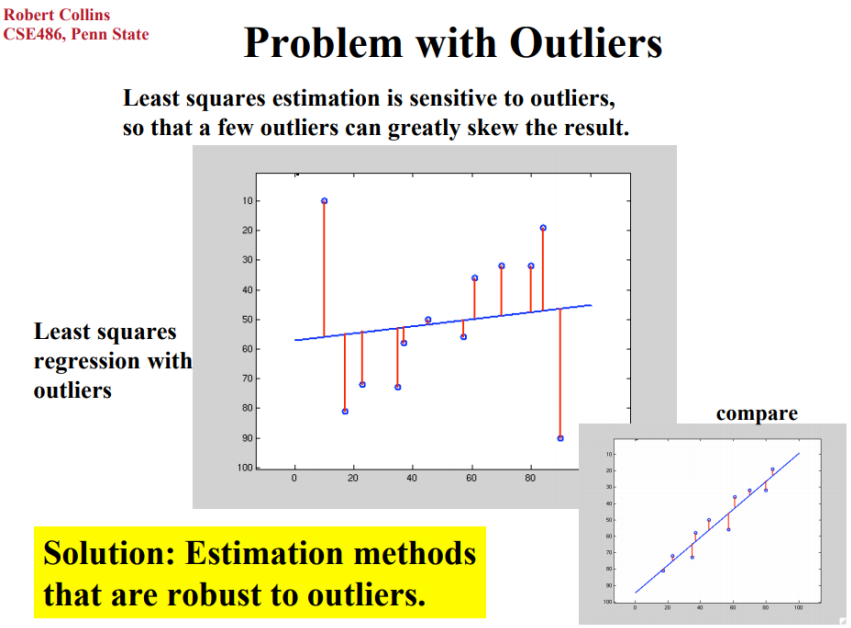
```
# PassThrough filter
# Create a PassThrough filter object
passthrough = cloud_vox_filtered.make_passthrough_filter()
#Assign axis and range to the passthrough filter object
filter_axis = 'z'
passthrough.set_filter_field_name(filter_axis)
axis_min = 0.55
axis_max = 0.75
passthrough.set_filter_limits(axis_min, axis_max)
#Use the filter function to obtain the filtered cloud

cloud_filtered = passthrough.filter()
```

RANdom Sample And Consensus (RANSAC) quickly identifies a set of points that most accurately fits a plane. The tools for RANSAC are included within the Point Cloud Library. A small set of bindings to the C++ library were made available within Python. Utilizing this, a plane is applied to identify the plane of the table surface. Points within a certain

distance of the plane are assumed be a part of the table. The remaining points are assumed to be a objects.

```
    # RANSAC plane segmentation
    # Create segmentation of the 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)
    # Maximum distance for a point to be considered fitting the model
    max_distance = 0.01
    seg.set_distance_threshold(max_distance)
    # Call the segment function to obtain set of inlier indicies and
model coefficients
    inliers, coefficients = seg1.segment()
    extracted_inliers = cloud_filtered.extract(inliers,
negative=True)
    extracted_outliers = cloud_filtered.extract(inliers,
negative=False)
```

RANSAC is used because it is robust in data fitting. Least squares methods can provide poor results.



Two clouds are created. Extracted inliers and Extracted outliers.

Exercise 2

Exercise 2 builds upon the foundation of Exercise 1 and creates clusters of objects with the points separated from the table. A cloud of XYZ data (no RGB information) is passed to a KD tree. Any point may be applied to a cluster as long as its within 0.05 units from the next object. At least 50 points are required to make a cluster, and no more than 3000 points should be included in that cluster.

```
    # Apply function to convert XYZRGB to XYZ
    white_cloud = XYZRGB_to_XYZ(extracted_inliers)
    # Create a K-D Tree
    tree = white_cloud.make_kdtree()
    # 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)
    #
    ec.set_ClusterTolerance(0.05)
    ec.set_MinClusterSize(50)
    ec.set_MaxClusterSize(3000)
    # Search the k-d tree for clusters
    ec.set_SearchMethod(tree)
    # Extract indices for each of the discovered clusters
    cluster_indices = ec.Extract()
```

Colors are assigned to each cluster to visualize the results.

```
    # Assign a color corresponding to each segmented object in scene
    cluster_color = get_color_list(len(cluster_indices))
    color_cluster_point_list = []
    for j, indices in enumerate(cluster_indices):
        for i, indice in enumerate(indices):
            color_cluster_point_list.append([white_cloud[indice][0],

      white_cloud[indice][1],

      white_cloud[indice][2],

      rgb_to_float(cluster_color[j]) ])
    # Create new cloud containing all clusters, each with unique
color
    cluster_cloud = pcl.PointCloud_PointXYZRGB()
    cluster_cloud.from_list(color_cluster_point_list)
```

Utilizing pcl_helper.py, the objects and the table are published as ROS messages.

Exercise 3 – Object Recognition with Python, ROS, and PCL

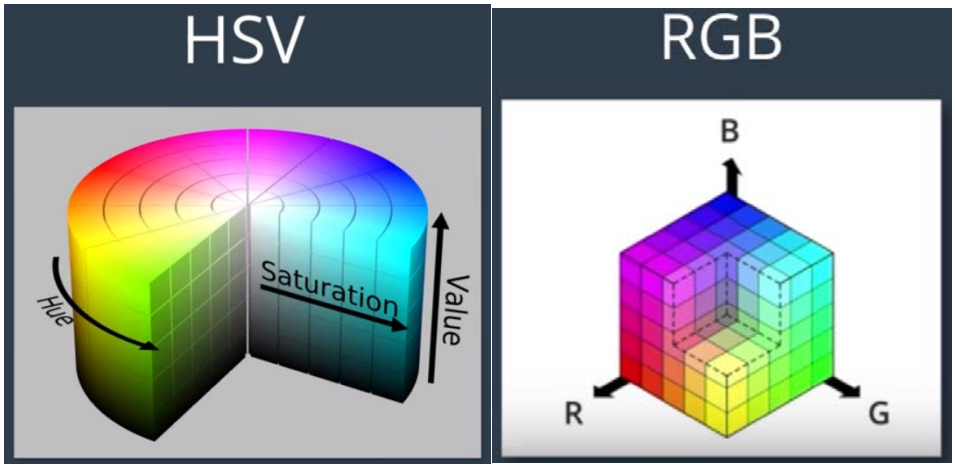
Exercise 3 continues in the same vein as Exercise 1 and 2. The previous tools are utilized and the data is passed into a new virtual environment. An assortment of objects are loaded in random angles at a given position in front of the depth camera.

Histograms are generated for the RGB values and for the surface normals of the objects. The RGB values are distributed into 32 separate bins across a range of 256 values. RGB values are defined from 0 to 255.

```
def compute_color_histograms(cloud, using_hsv=False):

    # Compute histograms for the clusters
    point_colors_list = []

    # Step through each point in the point cloud
    for point in pc2.read_points(cloud, skip_nans=True):
        rgb_list = float_to_rgb(point[3])
        if using_hsv:
            point_colors_list.append(rgb_to_hsv(rgb_list) * 255)
        else:
            point_colors_list.append(rgb_list)

    # Populate lists with color values
    channel_1_vals = []
    channel_2_vals = []
    channel_3_vals = []

    for color in point_colors_list:
        channel_1_vals.append(color[0])
        channel_2_vals.append(color[1])
        channel_3_vals.append(color[2])

    # TODO: Compute histograms
    r_hist = np.histogram(channel_1_vals, bins=32, range=(0, 256))
    g_hist = np.histogram(channel_2_vals, bins=32, range=(0, 256))
    b_hist = np.histogram(channel_3_vals, bins=32, range=(0, 256))
    # TODO: Concatenate and normalize the histograms
    hist_features = np.concatenate((r_hist[0], g_hist[0],
b_hist[0])).astype(np.float64)
    norm_features = hist_features / np.sum(hist_features)

    # Generate random features for demo mode.
    # Replace normed_features with your feature vector
    #normed_features = np.random.random(96)
    #return normed_features

    return norm_features
```

RGB are sensitive to lighting conditions and can provide poor results for matching. Different color spaces are available to compare items. Hue Saturation Value (HSV) is popular as it is less sensitive to lighting conditions. L*a*b* is also useful with more

sensitivity to what's considered "human perception."



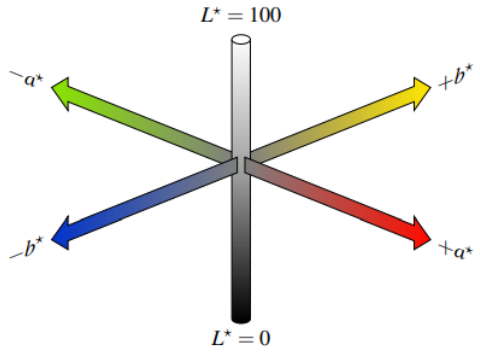


Figure 3: Illustration of the L*a*b* color space.

Histograms for surface normal must also be calculated. For consistency, 32 bins are used again but this time in a range of 0 to 1. Normals are unit vectors and should not exceed 1. The function is extremely similar to the RGB function.

```
def compute_normal_histograms(normal_cloud):
    norm_x_vals = []
    norm_y_vals = []
    norm_z_vals = []

    for norm_component in pc2.read_points(normal_cloud,
                                          field_names = ('normal_x',
'normal_y', 'normal_z'),
                                          skip_nans=True):
        norm_x_vals.append(norm_component[0])
        norm_y_vals.append(norm_component[1])
        norm_z_vals.append(norm_component[2])

    # TODO: Compute histograms of normal values (just like with
color)
    norm_x_hist = np.histogram(norm_x_vals, bins=32, range=(0, 1))
    norm_y_hist = np.histogram(norm_y_vals, bins=32, range=(0, 1))
    norm_z_hist = np.histogram(norm_z_vals, bins=32, range=(0, 1))

    # TODO: Concatenate and normalize the histograms
    hist_features = np.concatenate((norm_x_hist[0], norm_y_hist[0],
norm_z_hist[0])).astype(np.float64)
    norm_features = hist_features / np.sum(hist_features)

    # Generate random features for demo mode.
    # Replace normed_features with your feature vector
    #normed_features = np.random.random(96)

    #return normed_features

    return norm_features
```

Project Pipeline

MODEL TRAINING

```
roslaunch sensor_stick training.launch
```

To train the system run the following sequence of commands

Roslaunch sensor_stick training.launch

Rosrun sensor_stick capture_features.py

Rosrun sensor_stick train_svm.py

The first command launches a virtual environment. The second command exports a training set of data. The third command trains object names to their features.

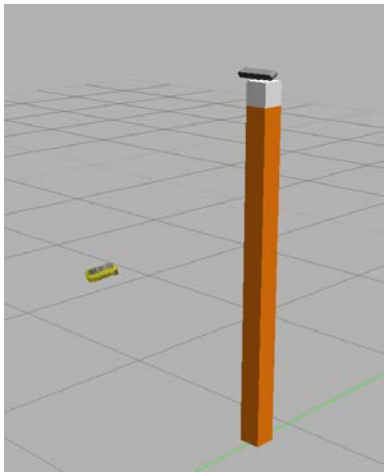


Figure 1: Random orientation of objects for data generation

Accuracy

Increasing the number of samples for training improves the capability of classifier across the features of HSV and normal. The value of more training samples decreases

exponentially, as the classifier performs with more and more precision.

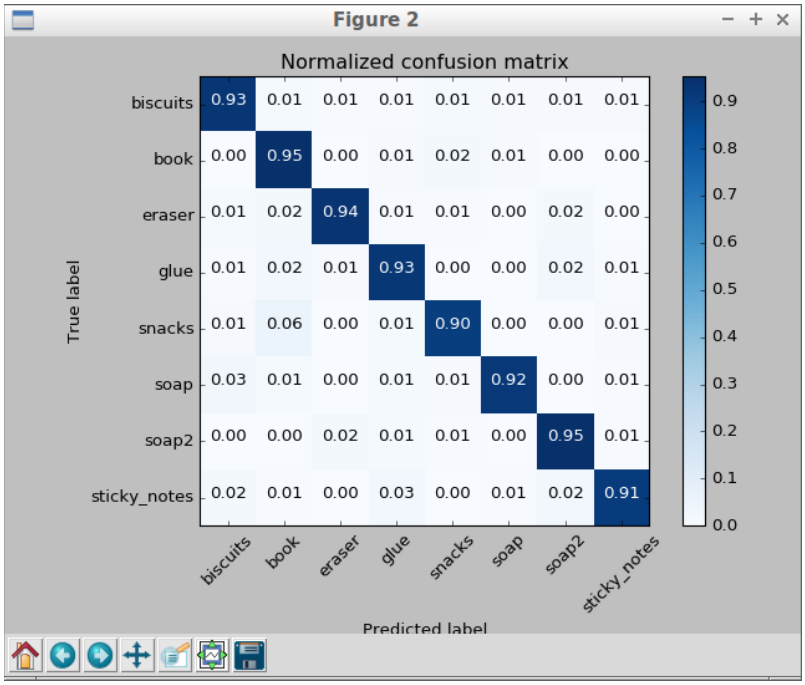
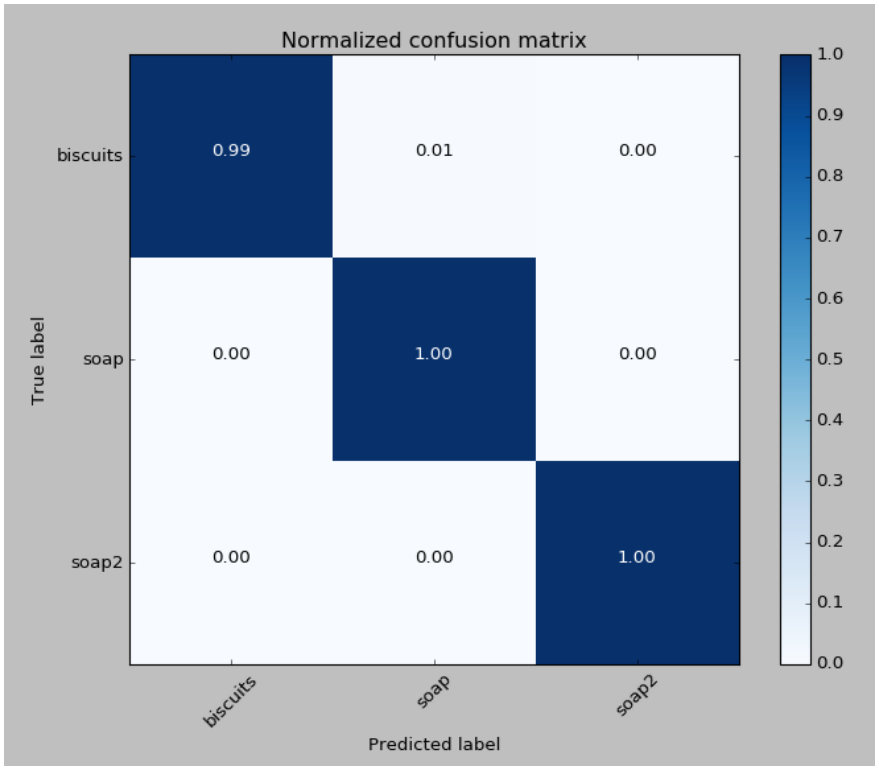


Figure 2: Confusion Matrix with HSV and Normals as features; 400 training samples

Since each pick list challenge is separate, separate SVMs are trained for each challenge.



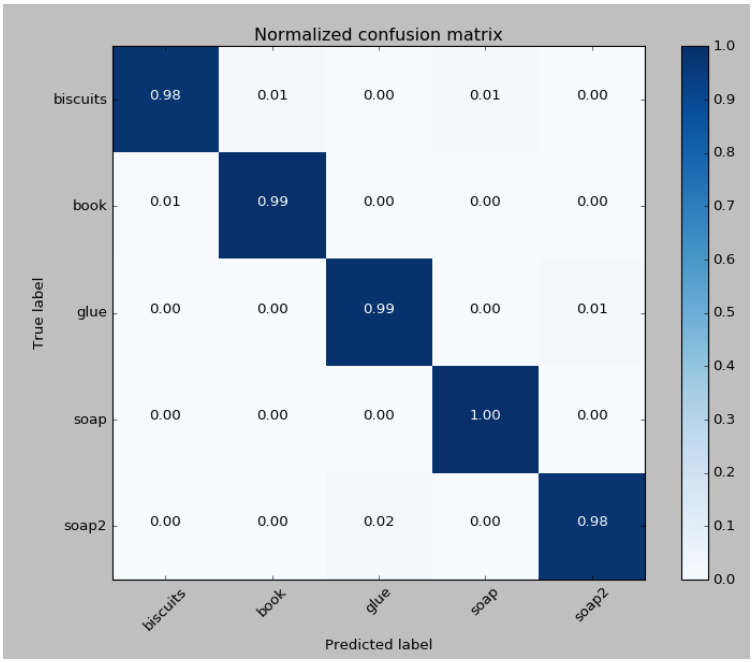


Figure 3: Training on Set 2 : 500 cycles

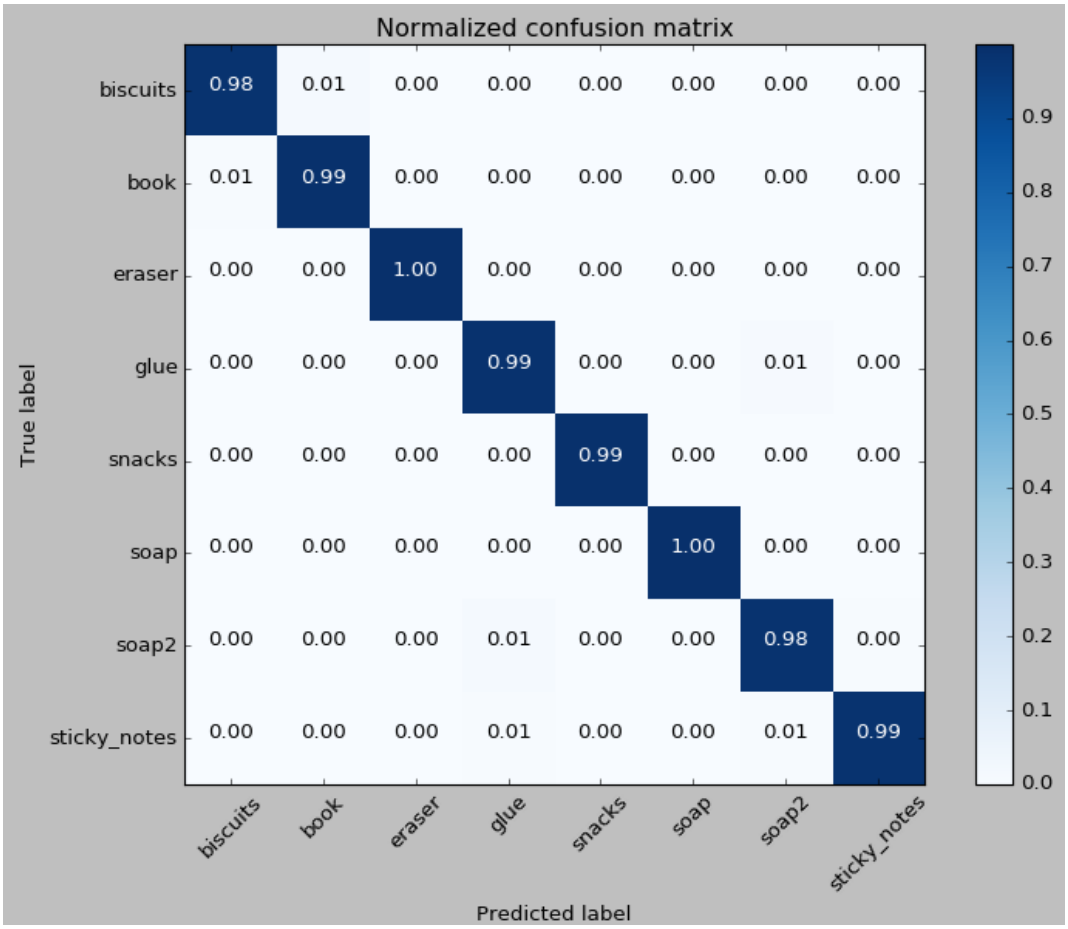


Figure 4: World 3 Trained on 1000 Cycles

Project Performance

LAUNCH

The environment is launched with the following command:

roslaunch pr2_robot pick_place_project.launch

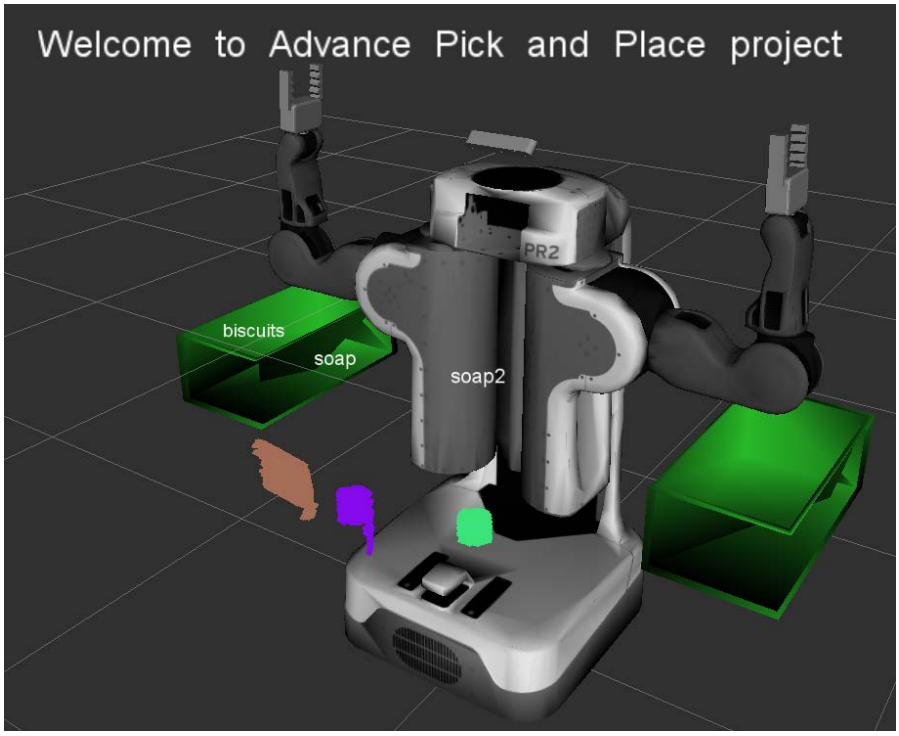
And the perception pipeline is launched with:

rosrun pr2_robot project_template.py

The different pick lists are identified by manipulating pr2_robot/launch/pick_place_project.launch. The variables test{}.world and pick_list_{}.yaml are modified based off of condition 1, 2, and 3.

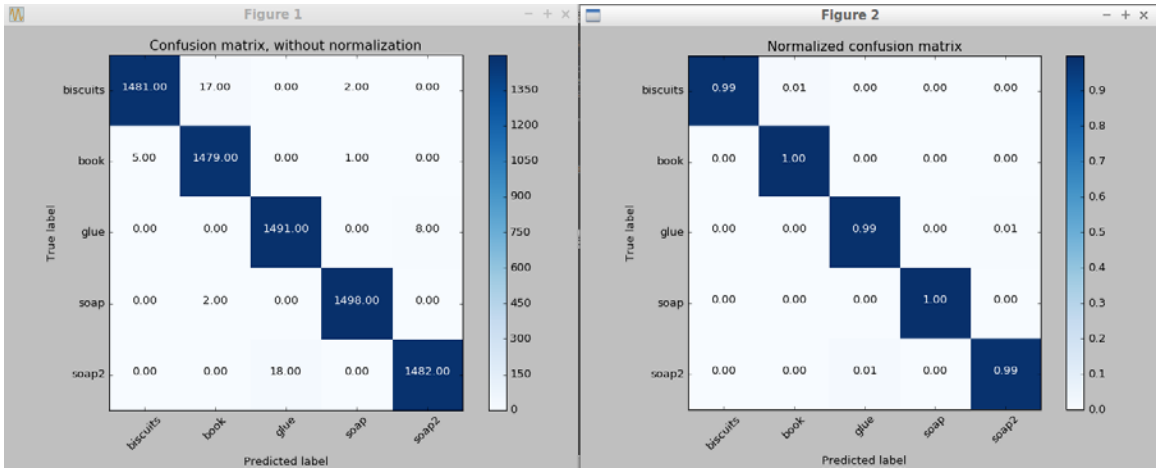
PICK LIST 1

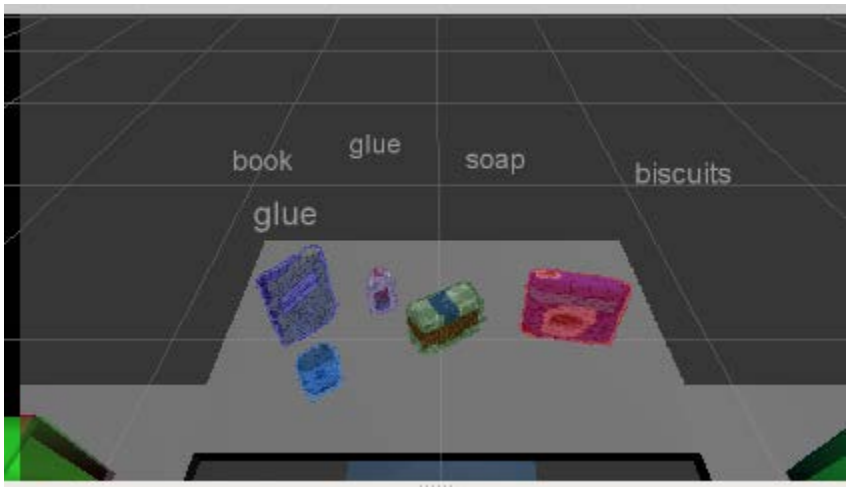
Pick list 1 loads biscuits, soap, and soap2 going into bins green, green, and red respectively. These were easily separated due to their large variance in hue.

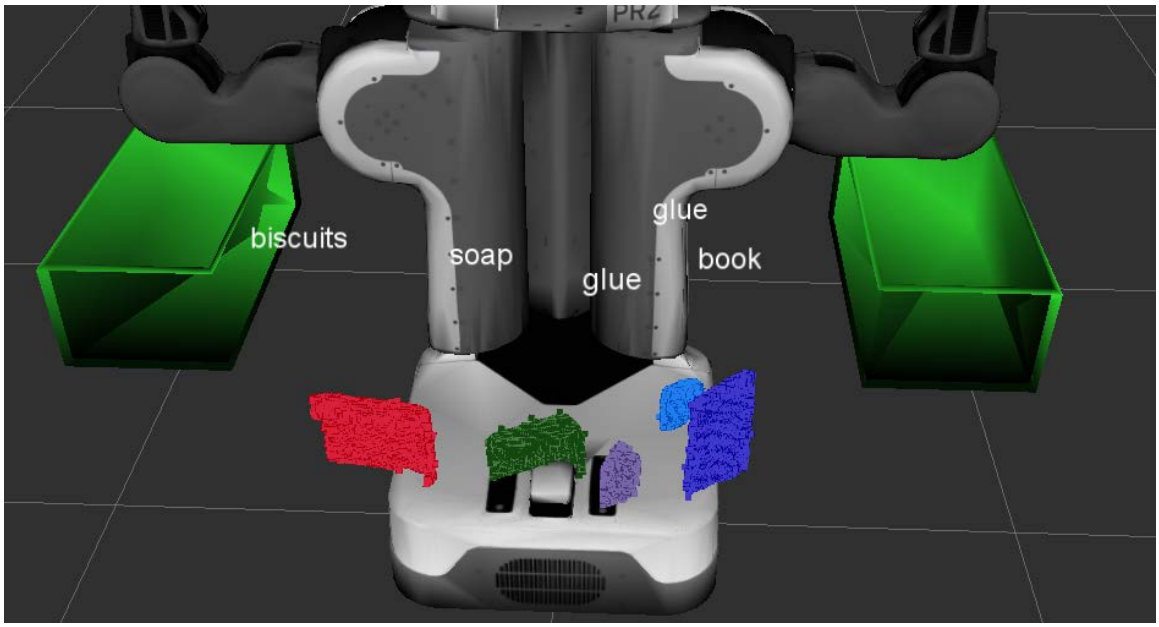


PICK LIST 2

The yaml for pick list 2 loads biscuits, soap, book, soap2, and glue. Going into bins green, green, red, red, and red respectively.



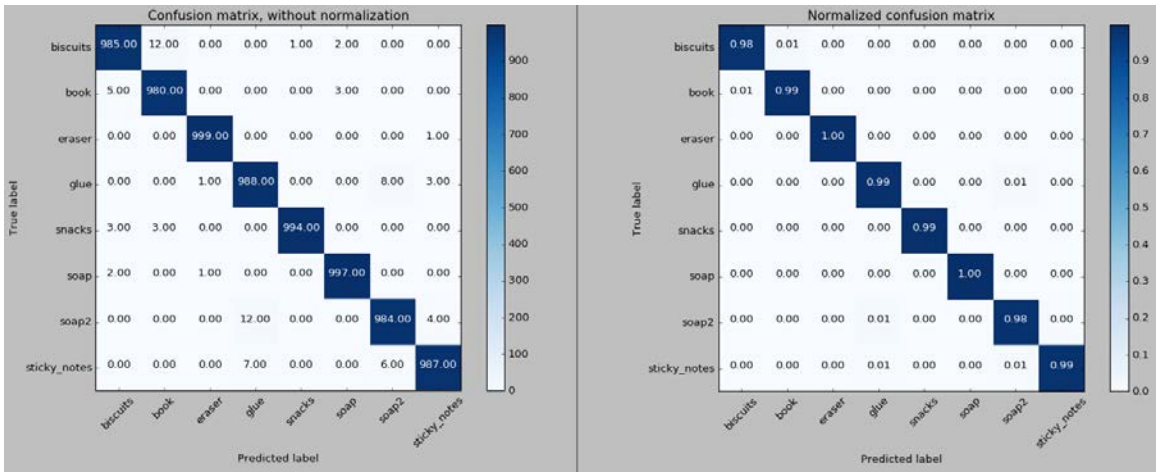


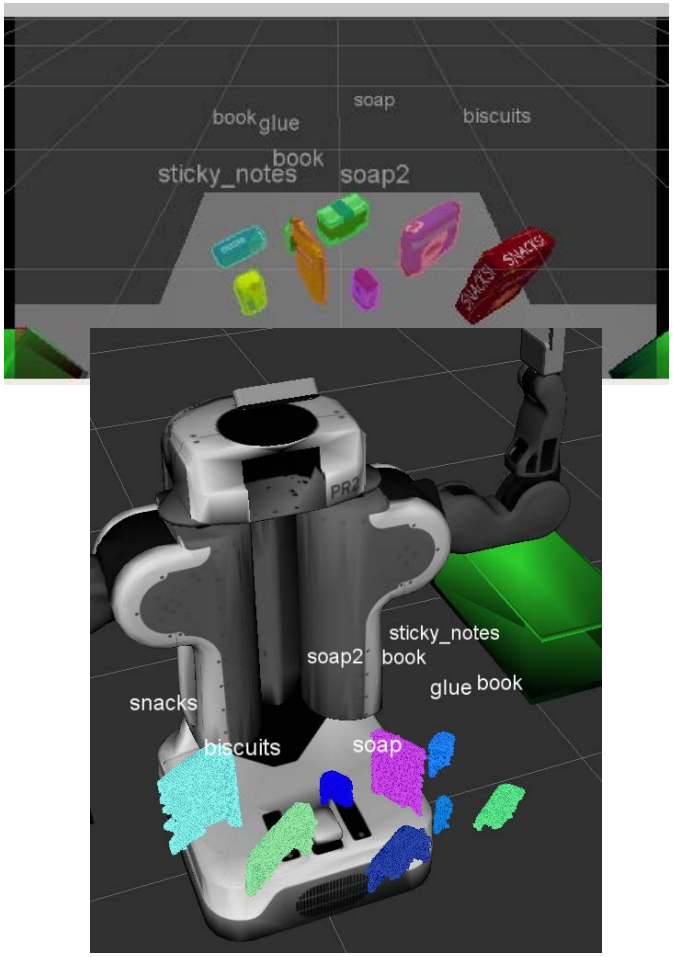


Here we can see we did not identify all of the objects correctly, as there are two “glue” items. Under conditions with less training, sometimes the book and biscuits are confused.

PICK LIST 3

The yaml for pick list 3 includes sticky_notes, book, snacks, biscuits, eraser, soap2, soap, and glue. The “snacks” is larger than any other item and did not get identified immediately due to too small of a cluster size. This had to be updated to prevent the cluster from being ignored.





At first “snacks” were not identified due to a clustering limit.

And then it was fixed here.

CONCLUSION

This project reinforces the concept that more data improves classifiers. Support Vector machines successfully identify various objects. The classifiers are only as good as the information provided to them. RGB color space was transitioned to HSV to better discriminate between objects, which led to learning about L*a*b* color space. Additional information could be used to improve the classifiers. In this project I added size information to the classifiers, but more advanced methods could include OCR, or neural networks to identify additional features.