Project: 3D Perception

In this project, I've identified objects on the basis of clustering, segmentation and object recognition techniques. Each of the individual techniques are clearly defined with pictures and code explanations.

- ## Exercise 1, 2 and 3 Pipeline Implemented

***Criteria 1:* Complete Exercise 1 steps. Pipeline for filtering and RANSAC plane fitting implemented.**

Path: Please check this path to find the code for this module : '/RoboND-Perception-Exercises/Exercise-1/RANSAC.py'

Solution: In this exercise, I've implemented RANSAC plane fitting technique. Alongside RANSAC, Voxel Grid Filter and Pass through Filter are also implemented. The steps for this exercise are as follows:

1. A point cloud data of a given table top scenario is taken as input.
2. The point cloud data is then converted into voxels of leaf size= 0.01 using Voxel Grid Filter. The leaf size 0.01 is taken considering the given scenario where the smallest object size is not beyond 0.01, therefore considering an even lower value will not provide any additional information. Similarly a higher value may lead to loss of information. Hence after experimenting 0.01 is considered as the most apt value for voxels. Fig 1, shows the output of this step.

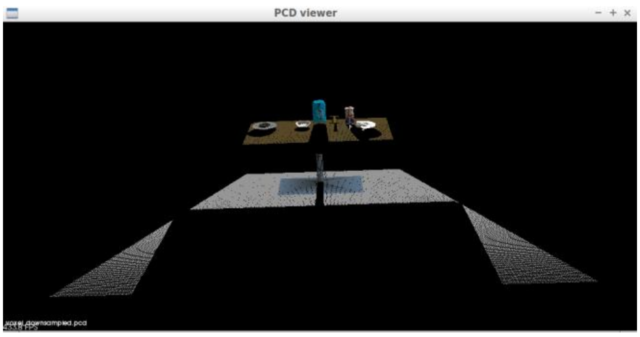


Fig 1 : Voxel Down-sampling of the point cloud for a table top scenario

```
# Load Point Cloud file
cloud = pcl.load_XYZRGB('tabletop.pcd')


## Voxel Grid filter

# Create a VoxelGrid filter object for our input point cloud
vox = cloud.make_voxel_grid_filter()


# Choose a voxel (also known as leaf) size
LEAF_SIZE = 0.01

# Set the voxel (or leaf) size
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)

# Call the filter function to obtain the resultant downsampled point cloud
cloud_filtered = vox.filter()
filename = 'voxel_downsampled.pcd'
pcl.save(cloud_filtered, filename)
```

Script 1 : Voxel Down-sampling of the point cloud for a table top scenario

3. The output of the previous step is then fed to Pass Through Filter, where from the voxelized point cloud data, only a particular Region of Interest(ROI) is filtered out. Along the z-axis in this table top scenario, we only need the table and the objects, so only the point cloud from a defined minimum z value to a defined maximum z value is required. Here in the code, I've

provided the z values as 0.77-1.1, the output after this filtering extracts only the point cloud of table and the objects:

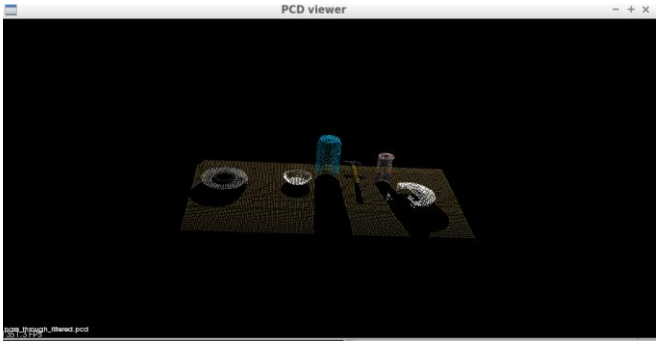


Fig 2 : Pass through Filter on the downsampled point cloud to extract the ROI (table and objects)

```
## PassThrough filter

# 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 = 0.77
axis_max = 1.1
passthrough.set_filter_limits(axis_min, axis_max)

# Finally use the filter function to obtain the resultant point cloud.
cloud_filtered = passthrough.filter()
filename = 'pass_through_filtered.pcd'
pcl.save(cloud_filtered, filename)
```

Script 2 : Pass through Filter Implementation

4. The output point cloud of Pass through filter is then allowed for plane fitting. As the entire table is on one plane, we can use a suitable model to extract the table point cloud from the rest of the point cloud. Here the table is on a x-y plane, with a constant z. The width of table, and the maximum distance till which the points of the table are allowed to fit the RANSAC model is considered as 0.01 on the basis of some experiments with the data. This separates the table from the rest of the objects. Hence from the point cloud after the extraction of Table, whatever is left out is our objects:

Fig 3 : RANSAC Plane Fitting Output with the extracted Plane and rest of the objects

Note: The table visibility in the figure above is not good, that’s why I’ve highlighted the table area, with a red boundary, a closer look to that area will show the table extracted.

```
## RANSAC plane segmentation

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

# RANSAC model to fit the plane
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
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()

# Extract inliers
extracted_inliers = cloud_filtered.extract(inliers, negative=False)
filename = 'extracted_inliers.pcd'
pcl.save(extracted_inliers, filename)                           # Save pcd for table

# Extract outliers
extracted_outliers = cloud_filtered.extract(inliers, negative=True)
filename = 'extracted_outliers.pcd'
pcl.save(extracted_outliers, filename)                          # Save pcd for tabletop objects

```

Script 3 : RANSAC Plane Fitting

***Criteria 2:* Complete Exercise 2 steps: Pipeline including clustering for segmentation implemented.**

Path: Please check this path to find the code for this module : <u>'/RoboND-Perception-Exercises/Exercise-2/sensor_stick/scripts/segmentation.py'</u>

Solution: In this exercise, I've implemented clustering of all the object point cloud data and then segmented them into individual clusters of different objects. After the steps of RANSAC.py in exercise 1, I've implemented the following steps:

5. The extracted point cloud data for the objects are taken as input in this step. This point cloud of objects is then clustered and segmented on the basis of distance between any 2 point cloud, the maximum number of point cloud that can exist in a cluster and the minimum number of points required to make a cluster. Euclidean Distance formula is used to obtain the clustering. For the given scenario, I've taken minimum number of points in cluster as 50, maximum number of points as 1700 and cluster tolerance as 0.02. This gives a neat picture of all the clusters for this table-top scenario in different colours:

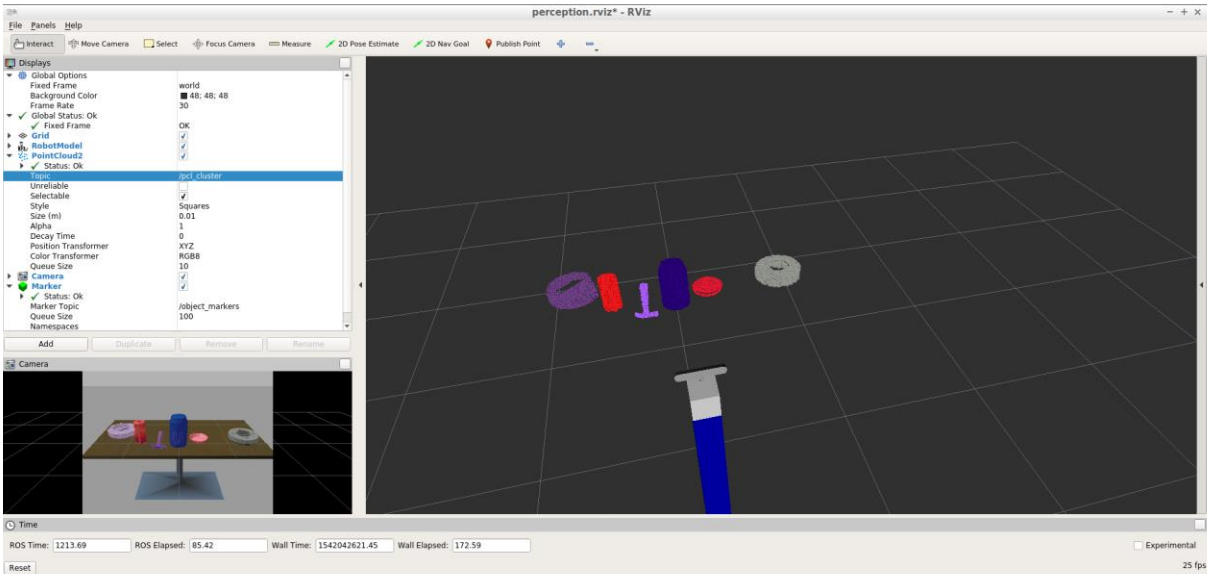


Fig 4 : Clustering and Segmentation of point cloud data into individual clusters shown in different colours

Note: The cluster list is given different random colours on the basis of the cluster length to visualize separate objects.

```
    # TODO: Euclidean Clustering
        white_cloud = XYZRGB_to_XYZ(extracted_outliers)                                    # Apply function to convert XYZRGB to XYZ
        tree = white_cloud.make_kdtree()

    # TODO: Create Cluster-Mask Point Cloud to visualize each cluster separately
                # 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(0.02)
        ec.set_MinClusterSize(50)
        ec.set_MaxClusterSize(1700)
        # Search the k-d tree for clusters
        ec.set_SearchMethod(tree)
        # Extract indices for each of the discovered clusters
        cluster_indices = ec.Extract()

        #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)
```

Script 4 : Segmentation Point Cloud On the basis of Euclidean Distance

6. The obtained clusters are converted into ROS messages which are then published to different publishers for objects, table and clusters. A ROS subscriber is created to subscribe to '/sensor_stick/point_cloud', and 'pcl_callback' function (having all the implemented steps mentioned above) is called within.

```
    # TODO: Convert PCL data to ROS messages
        ros_cloud_objects = pcl_to_ros(extracted_outliers)                    # Conversion to ROS messages for ALL Object Point Cloud
        ros_cloud_table = pcl_to_ros(extracted_inliers)                       # Conversion to ROS messages for Table Point Cloud
        ros_cluster_cloud = pcl_to_ros(cluster_cloud)                         # Conversion to ROS messages for INDIVIDUAL Object Point Cloud

    # TODO: Publish ROS messages
        pcl_objects_pub.publish(ros_cloud_objects)
        pcl_table_pub.publish(ros_cloud_table)
        pcl_cluster_pub.publish(ros_cluster_cloud)

if __name__ == '__main__':

    # TODO: ROS node initialization
        rospy.init_node('clustering', anonymous=True)

    # TODO: Create Subscribers
        pcl_sub = rospy.Subscriber("/sensor_stick/point_cloud", pc2.PointCloud2, pcl_callback, queue_size=1)

    # TODO: Create Publishers
        pcl_objects_pub = rospy.Publisher("/pcl_objects", PointCloud2, queue_size=1)
        pcl_table_pub = rospy.Publisher("/pcl_table", PointCloud2, queue_size=1)
        pcl_cluster_pub = rospy.Publisher("/pcl_cluster", PointCloud2, queue_size=1)

    # Initialize color_list
        get_color_list.color_list = []

    # TODO: Spin while node is not shutdown
        while not rospy.is_shutdown():
                rospy.spin()
```

Script 5 : ROS messages, subscribers and publishers for Point cloud, extracted table and individual Objects

Criteria 3:* *Complete Exercise 3 Steps. Features extracted and SVM trained. Object recognition implemented.

Path: Please check this path to find the code for this module : '/catkin_ws/src/sensor_stick/scripts/object_recognition.py'

Solution: In this exercise, I've implemented SVM on the table top objects with the help of colour histograms. 'training_setSS.sav', 'modelSS.sav' and 'capture_features.py' modules are used to represent this exercise. The steps are elaborated as follows:

Colour histograms: In the scenario given, the table top objects are: 'beer', 'bowl', 'create', 'disk_part', 'hammer', 'plastic_cup' and 'soda_can'. For all such objects, there are different colour properties. Every point cloud data for one particular object is having separate and unique RGB and HSV values. Here I've computed the HSV histogram of every point cloud object with nbins = 64 and the range as (0,256). This gives 64*64*64 Histogram features having values within the range of (0,256). These Histogram features are then normalized by the sum of total Histogram features. This I've named as 'compute_color_histograms' in 'features.py' code of 'sensor_stick' module. The same is repeated with normals of point cloud data as well, which is named as 'compute_normal_histograms' in 'features.py' code of 'sensor_stick' module.

```
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)
    nbins = 64
    bins_range=(0, 256)
            # Compute the histogram of the HSV channels separately
    h_hist = np.histogram(norm_x_vals, bins=nbins, range=bins_range)
    s_hist = np.histogram(norm_y_vals, bins=nbins, range=bins_range)
    v_hist = np.histogram(norm_z_vals, bins=nbins, range=bins_range)

    # TODO: Concatenate and normalize the histograms

            # Concatenate the histograms into a single feature vector
    hist_features = np.concatenate((h_hist[0], s_hist[0], v_hist[0])).astype(np.float64)
            # Normalize the result
    norm_features = hist_features / np.sum(hist_features)


    return norm_features
```

Script 6: Computation of Normal Color Histograms of point cloud data of every object

The obtained Color Histograms and Normal Histograms are the feature vectors of individual objects in the table top scenario, these features are then associated with the model name or the object name. Thus we obtained the labelled features of the table top objects.

In this code, to obtain the features of all the objects, I've allowed for 20 random orientations of every object. This gives me a better feature vector association of all the objects for the project scenarios with pick_list 1, 2, 3.

```
models = [\
   'beer',
   'bowl',
   'create',
   'disk_part',
   'hammer',
   'plastic_cup',
   'soda_can']

# Disable gravity and delete the ground plane
initial_setup()
labeled_features = []

for model_name in models:
    spawn_model(model_name)

    for i in range(20):
        # make five attempts to get a valid a point cloud then give up
        sample_was_good = False
        try_count = 0
        while not sample_was_good and try_count < 20:
            sample_cloud = capture_sample()
            sample_cloud_arr = ros_to_pcl(sample_cloud).to_array()

            # Check for invalid clouds.
            if sample_cloud_arr.shape[0] == 0:
                print('Invalid cloud detected')
                try_count += 1
            else:
                sample_was_good = True

        # Extract histogram features
        chists = compute_color_histograms(sample_cloud, using_hsv=False)
        normals = get_normals(sample_cloud)
        nhists = compute_normal_histograms(normals)
        feature = np.concatenate((chists, nhists))
        labeled_features.append([feature, model_name])

    delete_model()

pickle.dump(labeled_features, open('training_setSS.sav', 'wb'))
```

Script 7: Generating Training Set on the basis of Color Histograms

Train with SVM classifier: The training set obtained from the above mentioned steps, is then allowed to go into SVM. Here I've chosen the classifier as 'linear'. 'rbf ' classifier is also an option, but in most of my project scenarios and overall, 'linear' classifier gives me more accurate results when I compared in some experimental cycles.

The output of this training generates, the plots for ‘Confusion matrix, without normalization’ and ‘Normalized confusion matrix’ having the true labels and the predicted labels association among different objects of a given scenario. The normalized confusion matrix, for the given table top scenario is shown below:

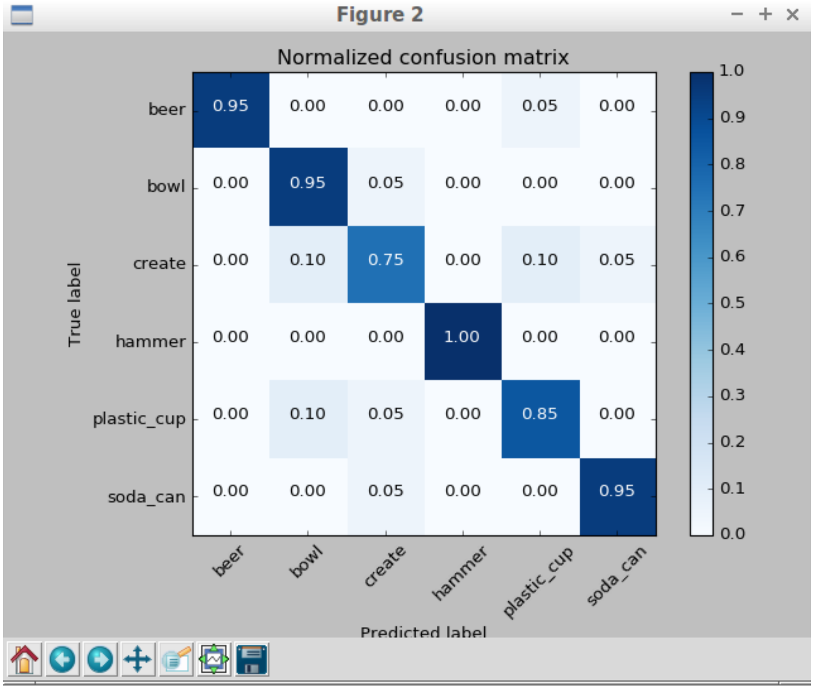


Fig 5 : Normalized Confusion Matrix for Table top Scenario

Here the association of the true labels and the predicted labels are verified with the diagonal values. Please note that, the minimum association is at 0.75 and overall it remains above 0.9. This generates the model set which is then allowed to recognize the objects in a ‘test set’.

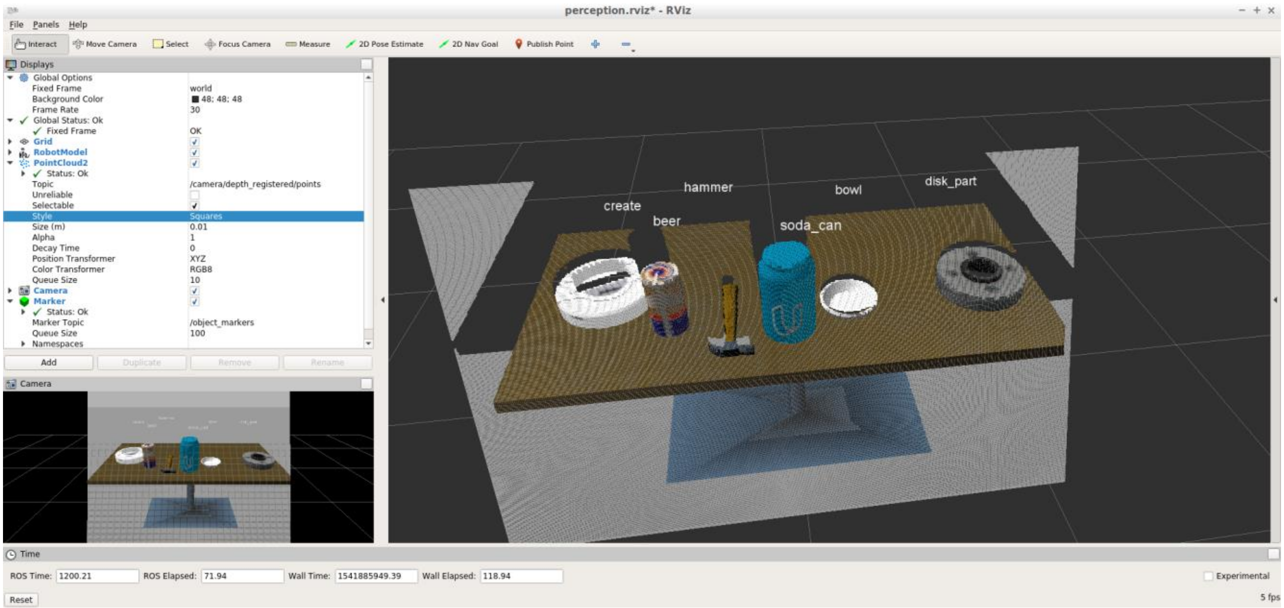
Object Recognition: This model set generated is then used for object recognition in actual scenario of a given test set. Next few steps are continued from Exercise 1 and 2:

7. The point cloud data from individual objects after Euclidean Clustering and segmentation is next allowed to go for colour based segmentation and association with the true labels of objects with a SVM classifier. The color histograms and normal histograms are calculated for the given test set to generate feature vector. This feature vector along with the model set generated while training the SVM, predicts and outputs the labels for the test set or the labels for the table top objects in this case.

```python
    # Initialize Variables
    detected_objects_labels = []
    detected_objects = []

    for index, pts_list in enumerate(cluster_indices):

            # Grab the points for the cluster from the extracted outliers (cloud_objects)
            cloud_objects = extracted_outliers
            pcl_cluster = cloud_objects.extract(pts_list)

            # TODO: convert the cluster from pcl to ROS using helper function
            ros_cloud_objects = pcl_to_ros(pcl_cluster)

            # Extract histogram features
            # TODO: complete this step just as is covered in capture_features.py
            chists = compute_color_histograms(ros_cloud_objects, using_hsv=True)
            normals = get_normals(ros_cloud_objects)
            nhists = compute_normal_histograms(normals)
            feature = np.concatenate((chists, nhists))

            # Make the prediction, retrieve the label for the result
            # and add it to detected_objects_labels list
            prediction = clf.predict(scaler.transform(feature.reshape(1,-1)))
            label = encoder.inverse_transform(prediction)[0]
            detected_objects_labels.append(label)

            # Publish a label into RViz
            label_pos = list(white_cloud[pts_list[0]])
            label_pos[2] += .4
            object_markers_pub.publish(make_label(label,label_pos, index))

            # Add the detected object to the list of detected objects.
            do = DetectedObject()
            do.label = label
            do.cloud = ros_cloud_objects
            detected_objects.append(do)

    rospy.loginfo('Detected {} objects: {}'.format(len(detected_objects_labels), detected_objects_labels))

        # Publish the list of detected objects

    detected_objects_pub.publish(detected_objects)
```

Script 8: Object Recognition with SVM and color histogram (HSV) properties of objects

The labels for this table top objects are then passed through ROS publishers to visualize the labels of objects on Rviz window:



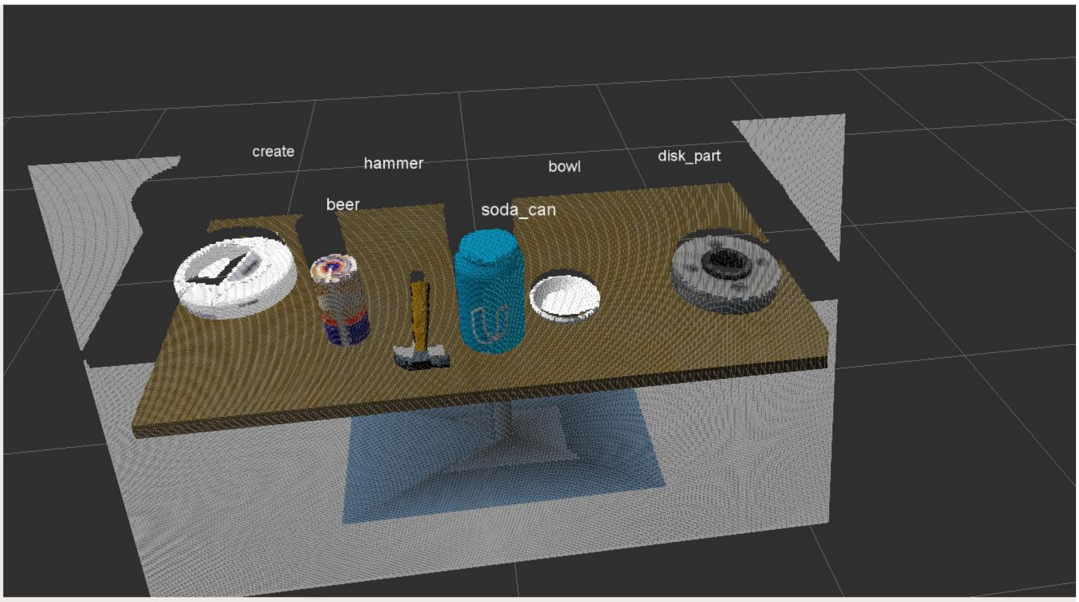


Fig 6 : Outputs of Object Labels after object recognition with 100% accuracy

➢ Pick and Place Setup

***Criteria 4:* For all three tabletop setups (test*.world), perform object recognition, then read in respective pick list (pick_list_*.yaml). Next construct the messages that would comprise a valid PickPlace request output them to .yaml format.**

Path: Please check this path to find the code for this module: '/RoboND-Perception-Project/pr2_robot/scripts/project_template.py'

Solution: In this Pick and place project, I've implemented the Clustering, Segmentation and classification following the steps in Exercise 1,2 and 3 with some new code for better implementation for the scenarios of 'test1.world, test2.world and test3.world'. After this I've generated yaml files for the respective scenarios named as 'output1.yaml, output2.yaml and output3.yaml'. The explanation of the code and results for passing the project is explained below:

1. In the 'project_template.py' code, the 'pcl_callback' function is used for point data clustering, segmentation and classification and 'pr2_mover' function is called to generate the pick and place request output to yaml files. The point cloud data is first generated from a ROS message, which is then allowed to pass through various filters as follows:

    a. *Statistical Outlier Filtering:* The point cloud data is noisy. To eliminate the noise, the outlier points are removed based on gaussian filtering, where the data points beyond the [mean + threshold_scale*standard deviation] value are removed from a cluster of a defined number of points. Here after a few iterations of experiments I've given the number of neighbouring points as 10 and threshold scale as 0.008.

```python
    # TODO: Convert ROS msg to PCL data
        pcl_data = ros_to_pcl(pcl_msg)

        cloud = pcl_data
    # TODO: Statistical Outlier Filtering

        # Much like the previous filters, we start by creating a filter object:
        outlier_filter = cloud.make_statistical_outlier_filter()

        # Set the number of neighboring points to analyze for any given point
        outlier_filter.set_mean_k(10)

        # Set threshold scale factor
        x = 0.008

        # Any point with a mean distance larger than global (mean distance+x*std_dev) will be considered outlier
        outlier_filter.set_std_dev_mul_thresh(x)

        # Finally call the filter function for magic
        cloud_filtered = outlier_filter.filter()
```

Script 9: Statistical Order Filtering

b. *Voxel Grid Downsampling:* The point cloud data after outlier removal is then converted into voxels of leaf size= 0.01 using Voxel Grid Filter. The leaf size 0.01 is taken, considering the given scenario, it could be different for different scenarios. Here although for this code, I've taken 0.01 as leaf size for all the test*.worlds.

```python
    # TODO: Voxel Grid Downsampling

        # Create a VoxelGrid filter object for our input point cloud
        vox = cloud_filtered.make_voxel_grid_filter()

        # Choose a voxel (also known as leaf) size

        LEAF_SIZE = 0.01

        # Set the voxel (or leaf) size
        vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)

        # Call the filter function to obtain the resultant downsampled point cloud
        cloud_filtered = vox.filter()

```

Script 10: Voxel Grid Downsampling

c. *Pass Through Filter:* In Pass Through Filter, from the voxelized point cloud data, only a particular Region of Interest(ROI) is filtered out. Along the z-axis for the scenarios from test*.worlds, the point cloud from a defined minimum z value to a defined maximum z value is extracted. Here in the code, I've provided the z values as 0.6-0.8, the output after this filtering extracts only the point cloud of table and the objects.

```python
    # TODO: PassThrough Filter
        # 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 = 0.6
        axis_max = 0.8
        passthrough.set_filter_limits(axis_min, axis_max)

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

```

Script 11: Pass Through Filter along z-axis

d. *RANSAC Plane Segmentation:* The output point cloud of Pass through filter is then allowed for plane fitting. As the entire table is on one plane, the table is extracted with a plane fitting model in RANSAC where the maximum distance allowed for the points to fit in the model is given as 0.009. This separates the table from the rest of the objects. Hence from the point cloud after the extraction of Table, whatever is left out is our objects. Here 'extracted_inliers' refers to the table and 'extracted_outliers' refers to the objects.

```
# TODO: RANSAC Plane Segmentation

    # 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

    max_distance = 0.009
    seg.set_distance_threshold(max_distance)

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

# TODO: Extract inliers and outliers
    extracted_inliers = cloud_filtered.extract(inliers, negative=False)

    # Extract outliers
    extract_outliers = cloud_filtered.extract(inliers, negative=True)
```

Script 12: RANSAC plane segmentation

After separating plane and objects, I observed that due to a wider FOV of Camera in pr2 robot, the edges of the bins are also coming as objects. To remove these as objects, I applied Pass through filter again to eliminate the points in y- direction based on a Region of Interest within [-0.5,0.5].

```
    # Extract outliers
    extract_outliers = cloud_filtered.extract(inliers, negative=True)

    passthrough = extract_outliers.make_passthrough_filter()            # Pass Through filter to remove the edges of the side boxes

    # Assign axis and range to the passthrough filter object.
    filter_axis = 'y'
    passthrough.set_filter_field_name(filter_axis)
    axis_min = -0.5                                                     # Along the y axis, objects are placed only on the front side of table i.e., within y axis [-0.5,0.5]
    axis_max = 0.5
    passthrough.set_filter_limits(axis_min, axis_max)

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

Script 13: Pass Through Filter along y-axis

e. *Euclidean Clustering:* The Objects point cloud or ‘extracted_outliers’ are then segmented into individual objects on the basis of distances amongst the point cloud data. For the given test*.worlds, I’ve taken minimum number of points in cluster as 50, maximum number of points as 1700 and cluster tolerance as 0.02. This gives a neat picture of all the clusters of different objects in different colours. This can be visualized in Rviz, with the help of pcl_cluster ROS message in publishers.

f. *PCL to ROS:* The point clouds for Table, all the objects and individual objects are converted from pcl to ros messages with the help of pcl_to_ros function from ‘pcl_helper.py’. These ros messages are then published to table, objects and cluster publishers. ROS subscribers are created for "/pr2/world/points", within which ‘pcl_callback’ function is called. The detected objects from all the ‘pick_list’ are published with object_markers on top of each object in Rviz window.

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

# TODO: Euclidean Clustering
    white_cloud = XYZRGB_to_XYZ(extracted_outliers)         # Apply function to convert XYZRGB to XYZ
    tree = white_cloud.make_kdtree()

# TODO: Create Cluster-Mask Point Cloud to visualize each cluster separately

    # 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(0.02)
    ec.set_MinClusterSize(50)
    ec.set_MaxClusterSize(1700)
    # Search the k-d tree for clusters
    ec.set_SearchMethod(tree)
    # Extract indices for each of the discovered clusters
    cluster_indices = ec.Extract()

    #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)
```

Script 14: Cluster Segmentation of all the pick_list objects

g. *Feature Extraction:* For every pick_list, the objects or the model names are modified in capture_features.py code. For pick_list_1.yaml, pick_list_2.yaml, pick_list_3.yaml, I've created capture_features1.py, capture_features2.py, capture_features3.py respectively with the model name as the objects given in individual pick_lists. This is also possible by just changing the models variable in one capture_features.py code, but I've chosen to create 3 for my ease.

For all the objects present in individual 'pick_list_*.yaml', corresponding 'capture_features*.py' creates a 'training_set*.sav' based on the color histograms computed. These 'training_set*.sav' is then allowed to train the SVM with a classifier 'linear', which generates the percentage of predicted labels with respect to their true labels and outputs 'model*.sav' to be used inside 'project_template.py ' code for a given test*.world.

```python
## Feature Extraction
    # Publish the list of detected objects
        for index, pts_list in enumerate(cluster_indices):

                # Grab the points for the cluster from the extracted outliers (cloud_objects)
                cloud_objects = extracted_outliers
                pcl_cluster = cloud_objects.extract(pts_list)

                # TODO: convert the cluster from pcl to ROS using helper function
                ros_cloud_objects = pcl_to_ros(pcl_cluster)

                # Extract histogram features
                # TODO: complete this step just as is covered in capture_features.py
                chists = compute_color_histograms(ros_cloud_objects, using_hsv=True)
                normals = get_normals(ros_cloud_objects)
                nhists = compute_normal_histograms(normals)
                feature = np.concatenate((chists, nhists))

                # Make the prediction, retrieve the label for the result
                # and add it to detected_objects_labels list
                prediction = clf.predict(scaler.transform(feature.reshape(1,-1)))
                label = encoder.inverse_transform(prediction)[0]
                detected_objects_labels.append(label)

                # Publish a label into RViz
                label_pos = list(white_cloud[pts_list[0]])
                label_pos[2] += .2
                object_markers_pub.publish(make_label(label,label_pos, index))

                # Add the detected object to the list of detected objects.
                do = DetectedObject()
                do.label = label
                do.cloud = ros_cloud_objects
                detected_objects.append(do)

        rospy.loginfo('Detected {} objects: {}'.format(len(detected_objects_labels), detected_objects_labels))
        detected_objects_pub.publish(detected_objects)
```

Script 15: Feature Extraction with SVM and color histogram (HSV) properties of objects in pick_list_*.yaml for Object Recognition

The 3 test scenarios are changed through pick_place_project.launch:

```
<arg name="world_name" value="$(find pr2_robot)/worlds/test3.world"/>

<!--TODO:Change the list name based on the scene you have loaded-->
<rosparam command="load" file="$(find pr2_robot)/config/pick_list_3.yaml"/>
```

For the execution of object recognition, in capture_features*.py code, 20 random orientations of the objects are taken, with number of histogram bins as 64 and range as (0,256) with a 'linear' kernel for SVM to further train the objects.

Please follow the below information in case, you want to generate normalization matrix else proceed with the model*.sav generated by me in this path: 'RoboND-Perception-Project/pr2_robot/scripts' and run for all the test*.world just by changing the model number, output*.yaml number and test_scene_num. For E.g, for [test1.world: test_scene_num.data = 1, yaml_filename = 'output1.yaml' and model = pickle.load(open('model1.sav', 'rb'))]

[Path: Please find the associated capture_features1.py for test1.world, capture_features2.py for test2.world and capture_features3.py for test3.world in this path : '/sensor_stick/scripts/capture_features*.py'

To observe the Normalization matrix, please change the training_set and model as: 'training_set1.sav and model1.sav for test1.world', 'training_set2.sav and model2.sav for test2.world' and 'training_set3.sav and model3.sav for test3.world'.**]**

Results and Observations:

- ✓ **'test1.world' :** For test1.world, the pick_list_1.yaml includes: [object_list:- name: biscuits, group: green, - name: soap, group: green, - name: soap2, group: red] which means these object names should be recognized after Object Recognition and then for pick place service request, the placement for the respective objects should be in their respective color boxes.

After the capture_features1.py and train_svm.py, the normalized matrix from training the SVM classifier gives the output as:

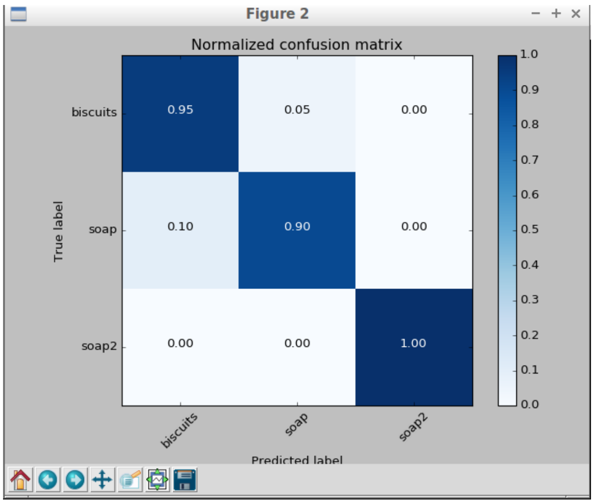


Fig 7 : Normalized Confusion Matrix for test1.world and pick_list_1.yaml

```
Features in Training Set: 60
Invalid Features in Training set: 0
Scores: [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.
  1.  1.  1.  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.]
Accuracy: 0.95 (+/- 0.44)
accuracy score: 0.95
```

Fig 8 : Accuracy Output in terminal for test1.world and pick_list_1.yaml

Analysis: For objects [biscuits, soap, soap2] the normalized value of predicted label remains above 0.9 with respect to their true labels. This suggests a good accuracy for objects in training set.This is then allowed to run in project_template.py code, in Gazebo and Rviz world. As shown in the figure below, all the objects are recognized with 100% accuracy in test1.world:

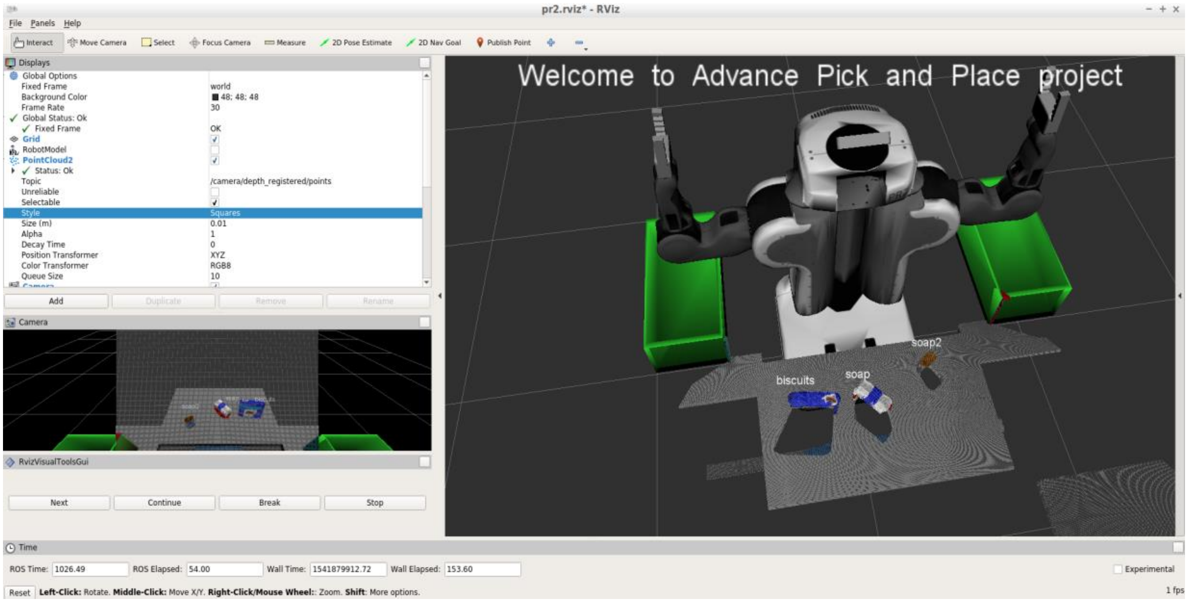


Fig 9 : Outputs of Object Labels after object recognition with 100% accuracy from pick_list_1.yaml

- ✓ **'test2.world' :** Similar to test1.world, SVM is trained with 'train_svm.py' code and 'capture_features2.py' code for 'pick_list_2.yaml' objects: [object_list: - name: biscuits, group: green, - name: soap, group: green, - name: book, group: red, - name: soap2, group: red, - name: glue, group: red] . The normalized matrix from training the SVM classifier gives the output as:

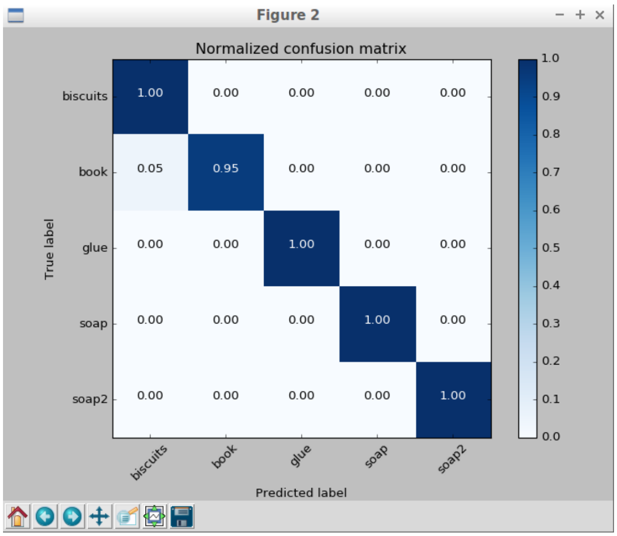


Fig 10 : Normalized Confusion Matrix for test2.world and pick_list_2.yaml

```
Features in Training Set: 100
Invalid Features in Training set: 0
Scores: [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
Accuracy: 0.99 (+/- 0.20)
accuracy score: 0.99
```

Fig 11 : Accuracy Output in terminal for test2.world and pick_list_2.yaml

Analysis: For objects [biscuits, book, glue, soap, soap2] the normalized value of predicted label remains above 0.95 with respect to their true labels. This suggests a good accuracy for objects in training set. Also the terminal shows the accuracy score of 0.99, which is again a very good estimate.

This is then allowed to run in project_template.py code, in Gazebo and Rviz world. As shown in the figure below, all the objects are recognized with 100% accuracy in test2.world for pick_list_2.yaml:

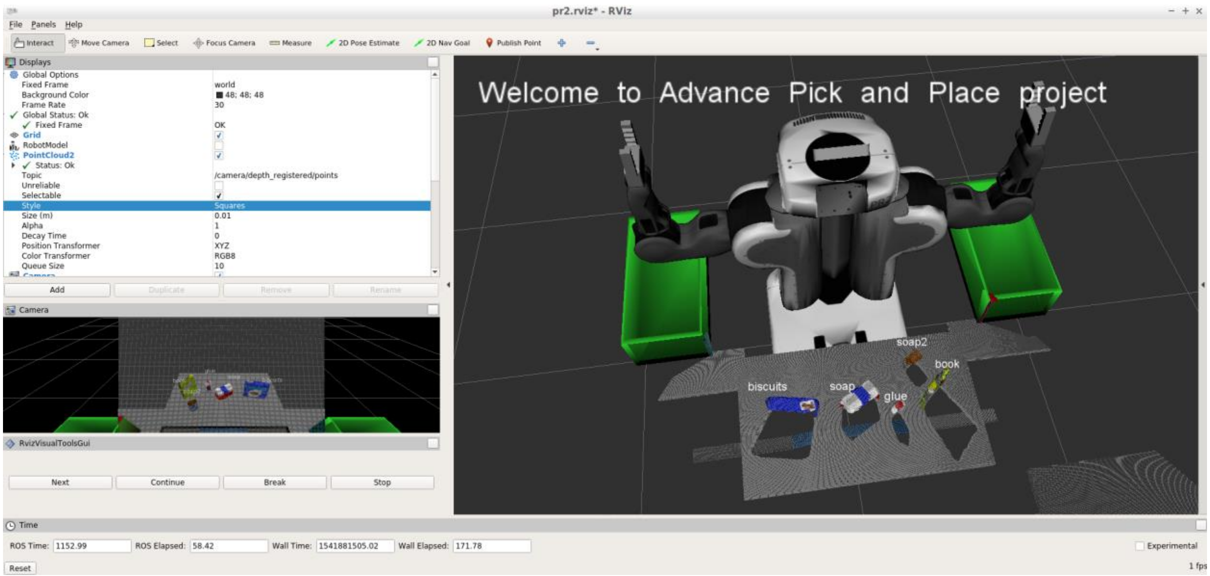


Fig 12 : Outputs of Object Labels after object recognition with 100% accuracy from pick_list_2.yaml

✓ **'test3.world' :** Similar to test1.world and test2.world, In test3.world, SVM is trained with 'train_svm.py' code and 'capture_features3.py' code for 'pick_list_3.yaml' having 8 objects: [object_list: - name: sticky_notes, group: red, - name: book, group: red, - name: snacks, group: green, - name: biscuits, group: green, - name: eraser, group: red, - name: soap2, group: green, - name: soap, group: green- name: glue, group: red]. The normalized matrix from training the SVM classifier for this test3.world gives the output as:

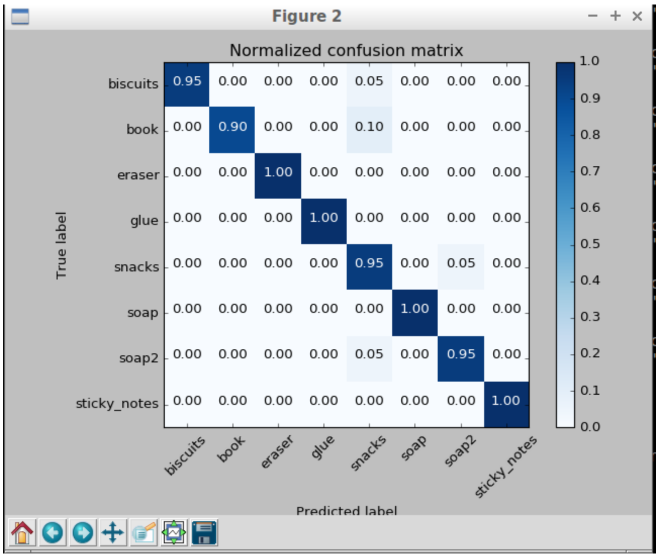


Fig 13 : Normalized Confusion Matrix for test3.world and pick_list_3.yaml

```
Features in Training Set: 160
Invalid Features in Training set: 0
Scores: [ 1.  1.  1.  1.  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
Accuracy: 0.97 (+/- 0.35)
accuracy score: 0.96875
```

Fig 14 : Accuracy Output in terminal for test3.world and pick_list_3.yaml

Analysis: For objects [biscuits, book, eraser, glue, snacks, soap, soap2, sticky_notes] the normalized value of predicted label remains around 0.9 or above with respect to their true labels. This suggests a good accuracy for objects in training set. Also the terminal shows the accuracy score of 0.96875, which is again a very good estimate.

This is then allowed to run in project_template.py code, in Gazebo and Rviz world. As shown in the figure below, all the objects are recognized with 100% accuracy in test3.world for pick_list_3.yaml:

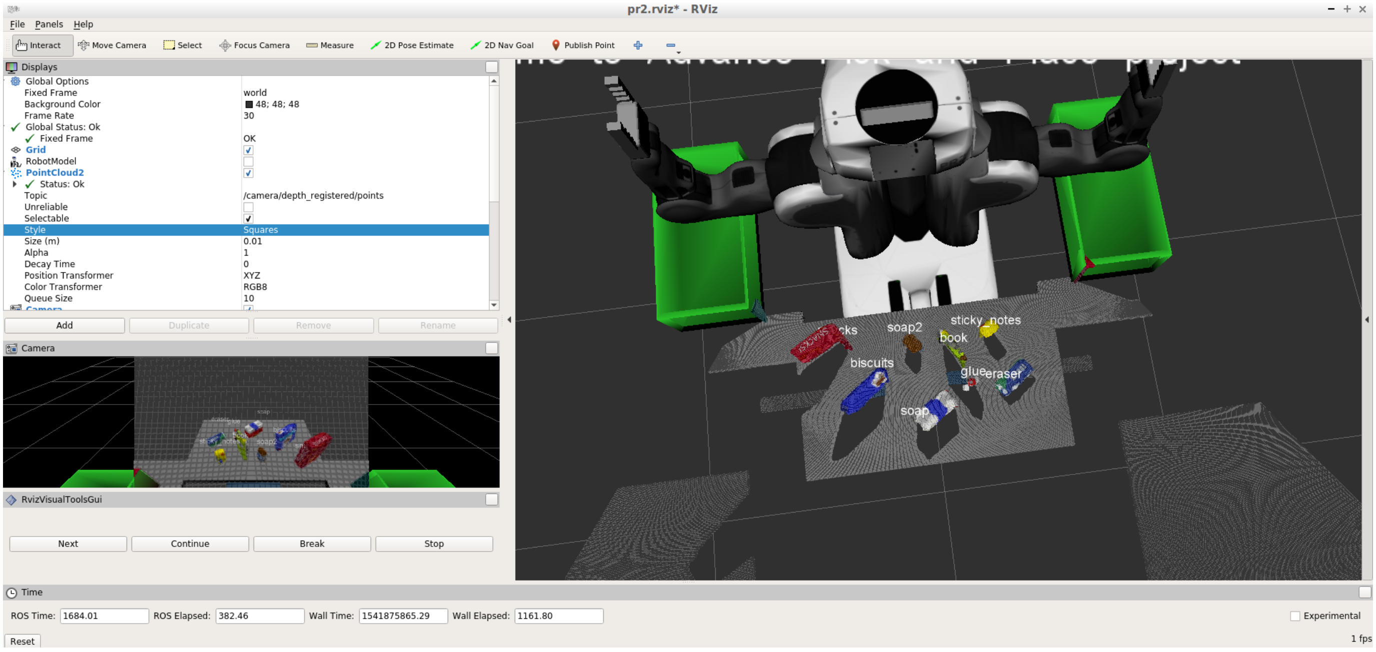


Fig 15 : Outputs of Object Labels after object recognition with 100% accuracy from pick_list_3.yaml

h. *pr2_mover Pick Place service:* Continuing from Object Recognition steps, there is requirement of obtaining .yaml files for individual pick_list which will specify: [test_scene_num, arm_name, object name, pick_pose, place_pose].

The algorithm logic for this is explained with screenshots of code as below. The test_scene_num corresponds to pick_list number.

```
        try:
                pr2_mover(detected_objects_labels,detected_objects)
        except rospy.ROSInterruptException:
                pass

# function to load parameters and request PickPlace service
def pr2_mover(object_list,detected_objects):

    # TODO: Initialize variables
        labels = [];                                    # Object Labels are stored here
        centroids = [];                                 # to be list of tuples (x, y, z)
        pick = [];                                      # Pick up positions (centroids) of all the object in terms of positions and orientations are stored
        place = [];                                     # Place positions of all the objects based on Left/Right bins in terms of positions and orientations are stored
        arm_name=[];                                    # List of arm names (Left/Right)
        object_name=[];                                 # List of object names
        OBJECT_NAME = [];                               # List of object names.data
        WHICH_ARM=[];                                   # List of arm names.data (Left/Right)
        pick_N = [];                                    # New Pick poitions after reordering the pick list names with the objects identified
        dict_list = [];                                 # Create dict_list of all the objects to output in yaml file

        # Test scene Number (Based on test*.world)
        test_scene_num = Int32()                        # Initialize a variable
        test_scene_num.data = 3                         # Populate the data field

```

Script 16: Calling pr2mover function for Pick Place service request

The object_name is the list of models present in the corresponding pick_list file. The arm_name is Right/Left based on color box Green/Red. The arm_name actually goes to the object closer to the box:

```python
# TODO: Get/Read parameters

    # get parameters
    object_list_param = rospy.get_param('/object_list')

# TODO: Parse parameters into individual variables

    # TODO: Assign the arm to be used for pick_place
    for i in range(0, len(object_list_param)):
            obj_name = String()                                                     # Initialize a variable
            obj_name.data=object_list_param[i]['name']                              # Populate the data field
            object_name.append(obj_name)
            OBJECT_NAME.append(obj_name.data)                                       # Extract the Object names fom the pick_list

            which_arm = String()                                                    # Initialize a variable
            which_arm.data = object_list_param[i]['group']                          # Populate the data field
            if which_arm.data=='green':
                    which_arm.data='right'                                          # If the group name for the object is green, assign the arm as Right
            else:
                    which_arm.data='left'                                           # If the group name for the object is red, assign the arm as Left
            arm_name.append(which_arm)
            WHICH_ARM.append(which_arm.data)
```

Script 17: Assigning object_name and arm_name based on group for every object in pick_list_*.yaml

The pick_pose is calculated with the help of centroids obtained from the point cloud cluster of separate objects.

The place_pose is calculated from 'dropbox.yaml' corresponding to the group (color) for each object in a given pick_list.yaml file. The dropbox_param[1] corresponds to Right arm and dropbox_param[0] corresponds to Left arm:

```python
# TODO: Get the PointCloud for a given object and obtain it's centroid

    for object in detected_objects:
            labels.append(object.label)
            points_arr = ros_to_pcl(object.cloud).to_array()
            c = np.mean(points_arr, axis=0)[:3]                                     # Calculate the centroids of the identified Point Cloud objects
            centroids.append(np.mean(points_arr, axis=0)[:3])
            pick_pose = Pose()                                                      # Initialize a variable

            pick_pose.position.x=np.asscalar(c[0])                                  # Populate the data field
            pick_pose.position.y=np.asscalar(c[1])
            pick_pose.position.z=np.asscalar(c[2])
            pick.append(pick_pose)


    for i in range(0, len(object_list_param)):                                      # Reorder the Pick positions of the objects,
            for j in range(0, len(labels)):                                         # based on the order in which the object list id defined in pick_list
                    if OBJECT_NAME[i] == labels[j]:
                            pick_N.append(pick[j])


    # TODO: Create 'place_pose' for the object
    dropbox_param = rospy.get_param('/dropbox')                                     # Extract place positions from dropbox.yaml

    for i in range(0, len(object_list_param)):
            place_pose = Pose()                                                     # Initialize a variable
            if WHICH_ARM[i]=='right':                                               # If the assigned arm is right, extract the place positions for Right
                    RightPos = dropbox_param[1]['position']
                    place_pose.position.x=RightPos[0]                               # Populate the data field
                    place_pose.position.y=RightPos[1]
                    place_pose.position.z=RightPos[2]
                    place.append(place_pose)

            else:                                                                   # If the assigned arm is left, extract the place positions for Left
                    LeftPos = dropbox_param[0]['position']                          # Populate the data field
                    place_pose.position.x=LeftPos[0]
                    place_pose.position.y=LeftPos[1]
                    place_pose.position.z=LeftPos[2]
                    place.append(place_pose)
```

Script 18: Assigning pick_pose and place_pose for every object in a given pick_list_*.yaml

After all the values for [test_scene_num, arm_name, object name, pick_pose, place_pose] are calculated for all the objects in a given pick_list_*.yaml, a dict_list is generated to combine all these values to be sent to output*.yaml file.

```python
        # TODO: Create a list of dictionaries (made with make_yaml_dict()) for later output to yaml format

        for i in range(0, len(object_list_param)):
        # Populate various ROS messages
                yaml_dict = make_yaml_dict(test_scene_num, arm_name[i], object_name[i], pick_N[i], place[i])
                dict_list.append(yaml_dict)

        # Wait for 'pick_place_routine' service to come up
        rospy.wait_for_service('pick_place_routine')

        try:
            pick_place_routine = rospy.ServiceProxy('pick_place_routine', PickPlace)

            # TODO: Insert your message variables to be sent as a service request
            #resp = pick_place_routine(test_scene_num, object_name[0], arm_name[0], pick[0], place[0])
            #print ("Response: ",resp.success)

        except rospy.ServiceException, e:
            print "Service call failed: %s"%e

    # TODO: Output your request parameters into output yaml file
        yaml_filename = 'output3.yaml'
        send_to_yaml(yaml_filename, dict_list)
```

Script 19: Generating dictionary list to output in output*.yaml file

```yaml
object_list:
- arm_name: left
  object_name: sticky_notes
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.4394531548023224
      y: 0.21640068292617798
      z: 0.6822409629821777
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0
      y: 0.71
      z: 0.605
  test_scene_num: 3
- arm_name: left
  object_name: book
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.49071958661079407
      y: 0.08390246331691742
      z: 0.7116703391075134
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
```

Fig 16 : Sample screenshot from output3.yaml

This completes the project and generates the output*.yaml files.

Path: Please check this path to find the output .yaml files : '/Outputs/output*.yaml'

Conclusion: In this project I've cleared the passing submission better than the requirements.

- My perception pipeline has correctly identified all the objects with 100% accuracy for test1.world, test2.world and test3.world as shown in Fig 9, Fig 12 and Fig 15. For the 3 given test*.world, its able to correctly predict and identify all the object labels.
- I've generated the output*.yaml files as per requirement with all the fields correctly filled. This is attached in the zip folder.
- I haven't taken the challenge for this project and therefore I'm not performing any pick and place operations. A few 'TODOs' corresponding to this and collision map in pr2_mover function has therefore been commented or left vacant for smooth running and generation of output*.yaml files.